Thijs Feryn

VARNISH 6

BY EXAMPLE

A practical guide to web acceleration and content delivery with Varnish 6 technology

VARNISH
SOFTWARE

Varnish 6 By Example
by Thijs Feryn
© 2021 Varnish Software AB
Layout: Tomas Arfert

Content

Foreword

Dear Reader,

First of all, thank you for deciding to take a look at the Varnish Book, whether in its digital or physical version. We'd say "read" the Varnish Book, but it is better aligned with our ambitions in writing the book to say "use" the book. We've written the book to make it as useful and efficient as possible by structuring it around practical examples because the potential use cases for Varnish technology can be very different.

For example, Varnish powers traditional website acceleration, scales and protects your video origin, works as an API gateway, delivers a full-blown, global, high-performance CDN, and takes care of 5G edge computing. As long as it is all about the HTTP protocol, Varnish has an important role to play whether you are looking for high-performance delivery, sub-millisecond latency, massive concurrency, or all of them combined.

Thijs Feryn has spent one year of his life creating this book. Thijs, being a Varnish evangelist, is usually a globetrotter, but due to Covid-19 he has been able to focus on putting together the most comprehensive description of Varnish technology to date. Thijs is the perfect person to write this book, not only because of his extensive Varnish experience, but also because he speaks to more Varnish users than anyone in the industry and never loses the customer and user perspective.

Thijs has been supported by a great team of Varnish core developers, editors and graphic designers to make sure the book delivers as much use to you as possible.

We hope you will agree.

Please let this book guide you, inspire you, and empower you.

Stockholm April 7, 2021

Lars Larsson, CEO of Varnish Software

Chapter 1: What is Varnish?

Varnish is industry-leading content delivery and edge computing software that speeds up websites, APIs, video streaming platforms, and content delivery platforms by leveraging uniquely powerful caching technology.

Varnish powers some of the world's most popular brands. It is used by millions of websites, including about 20% of the top 10,000 biggest websites worldwide, according to https://builtwith.com/.

In this chapter we'll go into further detail about the power of Varnish and the features that make it so powerful.

1.1 What is Varnish?

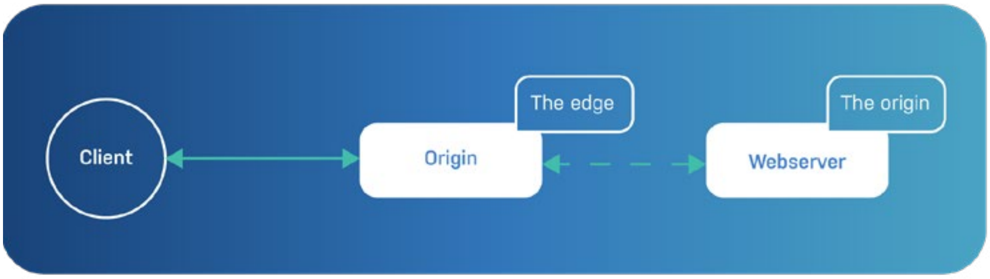
Originally, Varnish was a *reverse caching proxy*: a proxy server that speaks HTTP that you put in front of your web servers. Varnish heavily reduces the load and the latency of your web servers.

It does this by serving client requests with content that is cached in memory, eliminating the need to send each client request to the web servers.

However, when the content for a request is not available in cache, Varnish will connect to web servers to retrieve the requested content, and will attempt to store the response in cache for future requests.

By adding this new layer of caching, we divide the platform into two distinct tiers in terms of content delivery:

- *The origin:* represents your original web servers that are inherently prone to high load and latency, and that need to be protected in order to guarantee stability.
- *The edge:* the outer tier of your platform. It is secure, stable, fast and scalable. This is where users interact with your content and where Varnish really shines.



Basic Varnish Diagram

Because *Varnish* speaks HTTP and sits in front of the web servers, it seemingly assumes the role of the web server. The HTTP client that connects to the platform has no idea that Varnish is actually a proxy. In a lot of cases, the same applies to the origin servers: most of the time, they have no clue that Varnish is a proxy and not a regular HTTP client.

Varnish is available in two forms:

- The open source versions that we refer to as Varnish Cache
- The enterprise versions that we refer to as Varnish Enterprise

Varnish Enterprise is maintained by Varnish Software, whereas *Varnish Cache* is maintained by both *Varnish Software* and the open source community.

Varnish Software employs most of the engineers working on *Varnish Cache*. In addition *Varnish Software* maintains the *long-term support (LTS)* version of *Varnish Cache*.

More information about the differences between the two versions can be found in a dedicated section in this chapter.

In its default configuration, Varnish will respect `Cache-Control` headers from the web server, and cache objects for the amount of time the web server indicates, or not at all. There are built-in mechanisms to do this in a safe way, so that private information is not stored in the cache. This means that a web developer can gain a lot from Varnish with just some basic configuration.

However, much of the power of Varnish is that its behavior can be configured and changed in many ways. There are many parameters that can be tuned. Request handling and caching behavior can be altered, or completely redefined, using the *Varnish Configuration Language (VCL)*.

1.2 What is VCL?

VCL stands for *Varnish Configuration Language* and is the *domain-specific language* used by Varnish to control request handling, routing, caching, and several other things.

At first glance, VCL looks a lot like a normal, top-down programming language with subroutines, if-statements and function calls. However, it is impossible to execute *VCL* outside of Varnish. Instead, you write code that is run inside Varnish at specific stages of the request-handling process. This lets you define advanced logic that extends the default behavior of Varnish.

The locations where the *VCL* is run are actually different *states* of the *Varnish finite state machine*. Even though you can accomplish a lot by configuring just a few states, understanding the state machine is necessary to leverage the full potential of Varnish.

VCL is one of the most compelling Varnish features. It is often the main reason why users choose Varnish over competing web acceleration products. The level of flexibility that VCL offers is unparalleled in the industry.

Through a set of pre-defined subroutines and other language constructs in the *VCL file*, the behavior of Varnish can be extended. This can range from request and response header manipulation, to backend selection, overriding state transitions in the *finite state machine*, and many more other actions.

When the `varnishd` runtime process is started, the *VCL file* is processed, the *VCL code* is translated into C code, the C code is then compiled to a *shared object*, and eventually this shared object is linked to the server process, where its code is executed.

Here's some sample code to give you an idea of what VCL looks like:

```
vcl 4.1;

backend default {
    .host = "backend.example.com";
    .port = "80";
}

sub vcl_recv {
    if(req.url ~ "^/admin(/.*)?") {
        return(pass);
    }
}
```

This VCL snippet will instruct Varnish not to serve objects from cache if the URL is `/admin` or if the URL starts with `/admin/`. When a backend connection is made, Varnish will connect to the `backend.example.com` hostname on the conventional HTTP port, which is `80`.

Note that the *DNS resolution* of the hostname is done when the VCL configuration is loaded and not on every backend connection.

In addition to standard VCL, there's also a rich ecosystem of *VMODs*, or *Varnish modules*. These modules allow users to integrate with third-party C libraries, and add extra functionality to Varnish. A VMOD exposes its functionality through a set of functions and objects, which further enrich the VCL language.

Here's a VCL snippet that features the `cookie` VMOD:

```
vcl 4.1;
import cookie;

backend default {
    .host = "backend.example.com";
    .port = "80";
}

sub vcl_recv {
    cookie.parse(req.http.cookie);
    cookie.keep("language");
    set req.http.cookie = cookie.get_string();
    return(hash);
}

sub vcl_hash {
    hash_data(cookie.get("language"));
}
```

This VCL snippet will use the `cookie` VMOD to remove all incoming cookies except the `language` cookie. It will force a cache lookup, even when cookies are present. The value of the `language` cookie will be used as a cache variation to create a cache object per URL per language.

As you can see this VMOD adds additional functionality to Varnish and exposes this functionality using a VCL API.

We'll discuss VCL in much more detail in the dedicated VCL chapter.

1.3 Varnish Cache and Varnish Enterprise

As mentioned earlier, there are two versions of Varnish: an *open source* version and an *enterprise* version.

Originally, Varnish started out as tailor-made software for *Verdens Gang*, a Norwegian newspaper. The development of Varnish was spearheaded by long-time FreeBSD core contributor *Poul-Henning Kamp* in collaboration with Nordic open source service provider *Redpill Linpro*.

Eventually it was decided that the source code would be open sourced. Poul-Henning Kamp has remained the project lead, and continues to maintain *Varnish Cache* with the help of various people in the open source community and Varnish Software.

The success and enormous potential of the project led to *Varnish Software* being founded in 2010 as a spinoff of *Redpill Linpro*. Initially *Varnish Software* focused on support and training, which funded further development of the open source project.

In 2014, *Varnish Software* started developing specific features on top of *Varnish Cache* in a commercial version of the software, and named it *Varnish Plus*. It is now known as *Varnish Enterprise*.

The feature additions that *Varnish Enterprise* initially offered primarily consisted of extra *VMODs*, but as time went by, some substantial features were developed by *Varnish Software* that went beyond modules.

The most significant feature being *MSE*, which is short for *Massive Storage Engine*. *MSE* is a so-called *stevedore* that Varnish uses to store its cached objects. Unlike `malloc` *(memory storage stevedore)*, and `file` *(non-persistent disk storage stevedore)*, *MSE* offers a dual-layer storage solution that leverages the speed of memory, and the resilience of disk, without the typical slowdown effect of traditional disk-based storage systems.

In addition, *VHA*, which stands for *Varnish High Availability*, was introduced. This solution replicates stored cache objects across multiple Varnish servers.

Add built-in *client and backend TLS/SSL termination*, and a browser-based administration interface to that, and you have a pretty solid feature set.

The combination of these features, and the fact that they were shipped by default, supported and covered by a Service Level Agreement *(SLA)*, made *Varnish Enterprise* look pretty interesting to enterprise companies.

1.3.1 Version numbers

There is a correlation between the *Varnish Cache* and *Varnish Enterprise* version numbers.

Varnish Cache is on a six-month release schedule. Every year you'll see a release in March, and a release in September:

- On March 15th 2018 *Varnish Cache 6.0* was released
- On September 17th 2018 *Varnish Cache 6.1* was released
- On March 15th 2019 *Varnish Cache 6.2* was released
- On September 16th 2019 *Varnish Cache 6.3* was released
- On March 16th 2020 *Varnish Cache 6.4* was released
- On March 16th 2020 *Varnish Cache 6.4* was released
- On September 15th 2020 *Varnish Cache 6.5* was released
- On March 15th 2021 *Varnish Cache 6.6* was released

Varnish Enterprise 6 is based on *Varnish Cache 6.0*, and doesn't follow the minor version upgrades. Instead the *Varnish Software* team backports fixes and some of the features.

However, *Varnish Enterprise 6* does follow the patch version upgrades for *Varnish Cache 6.0*, but adds a release version number.

So when *Varnish Cache 6.0.1* was released on August 29th 2018, the corresponding *Varnish Enterprise* release was version *6.0.1r1*. When a new *Varnish Enterprise* release takes place, and there is no new *Varnish Cache 6.0* patch release; the release number just increases.

This happened for example on October first when *Varnish Enterprise 6.0.1r2* was released.

At the time of writing this book, the latest *Varnish Enterprise* version is *6.0.8r1*.

1.3.2 Product vs project

It would be too simplistic to conclude that *Varnish Enterprise* is just *Varnish Cache* with some extra features and an *SLA*. The differences are much more fundamental.

Varnish Cache is a project. *Varnish Enterprise* is a product.

This quote sums it up best, and they both have different goals.

Varnish Enterprise is a product that you install once, and then let run for several years, without having to put in a significant amount of effort every time a new minor version is released.

If for example you installed *Varnish Enterprise* version *6.0.1r1* when it came out back in 2018, and you now upgrade to *6.0.8r1*, everything will just work without any risk of incompatibility.

For *Varnish Cache*, the goal is to continuously improve the code and the architecture, and look toward the future. Compatibility breaks are discouraged, but every six months a release is cut, which might break users' setups. The *Varnish Cache* community tries to document everything that has changed enough to affect users. But it's still up to users to check whether or not these changes are compatible with their setup.

1.3.3 Which features does Varnish Cache have?

Varnish Cache is extremely fast and stable. It has a rich feature set that can be used:

- Out-of-the-box
- By writing VCL
- By leveraging some of the built-in Varnish Modules

Here's an overview of *Varnish Cache*'s features:

Feature	Description
Request coalescing	Protects origin servers against cache stampedes by collapsing similar requests
`Cache-Control` support	Varnish respects `Cache-Control` headers, and uses `max-age` and `s-maxage` values to define the object TTL. Directives like `public`, `private`, `no-cache`, `no-store`, and `stale-while-revalidate` are also interpreted.
`Expires` support	Varnish can interpret the `Expires` header and set the object TTL accordingly.
Conditional requests	Varnish supports *304 Not Modified* behavior by interpreting `ETag` and `Last-Modified` headers and issuing `If-None-Match` and `If-Modified-Since` headers. This is supported both at the client side and at the backend side.

Grace mode	Varnish's implementation of *Stale While Revalidate*. Varnish will serve stale objects while the latest version of the object is fetched in the background. The duration is configurable in VCL or via the `stale-while-revalidate` directive in the `Cache-Control` header.
Content streaming	Varnish will start streaming content to the client as soon as it has received the response headers from the backend.
Cache invalidation	Varnish has purging, banning, and content refresh capabilities to remove objects from cache.
LRU cache evictions	When the cache is full, Varnish will use a *Least Recently Used (LRU)* algorithm to remove the least recently used objects in an attempt to free up space.
HTTP/2 support	Varnish supports the HTTP/2 protocol.
Backend health checking	The health of a backend can be checked using configurable health probes. These checks can lead to backends not being selected for backend fetches.
Backend connection limiting	Limit the maximum number of open connections to a single backend.
Backend timeout control	Limit the amount of time Varnish waits for a valid back response. Configurable through various timeout settings
Advanced backend selection	Programmatically select the backend, based on a custom set of conditions written in VCL
Advanced request saving	Varnish can transparently save requests by retrying other backends if the initial backend request fail or serving stale content if the backend is unavailable.
Configurable listening addresses	Varnish can accept incoming HTTP requests on multiple listening addresses. Hostname/IP and port number are configurable
PROXY protocol support	Listening addresses can be configured to accept *PROXY protocol* requests rather than standard HTTP requests. The *PROXY PROTOCOL* will keep track of the IP address of the original client, regardless of the number of potential proxies in front of Varnish.
TLS termination	Although *Varnish Cache* doesn't support *TLS* natively, *TLS termination* can be facilitated with the *PROXY protocol*.

Unix domain socket support *(UDS)*	Both incoming connections and connections to backends can be made over *Unix domain sockets (UDS)* instead of *TCP/IP*
Stevedores	A *stevedore* is the storage mechanism that Varnish uses to store cached objects. `malloc` *(memory storage)* is the default. `file` *(non-persistent disk storage)* is also common
Command line interface *(CLI)*	Varnish has a *command line interface (CLI)* that can be used to tune parameters, ban objects from cache and load a new VCL configuration
Edge Side Includes *(ESI)*	XML-based placeholder tags whose `src` attributes are processed by Varnish *(on the edge)* and where the HTTP responses replace the placeholders.
Zero-impact config reload	Load a new VCL file without having to reload the Varnish process
Label-based multi-VCL configurations	Load multiple VCL files and conditionally execute them using labels in your main VCL file
Access control lists *(ACLs)*	Allow or restrict access to parts of your content using *access control lists (ACLs)*, containing IP addresses, hostnames or subnets that can be matched in VCL
URL transformation	Transform any URL in VCL
Header transformation	Transform any request or response header in VCL
Synthetic HTTP responses	Return custom HTTP responses that did not originate from your origin
VCL unit testing framework	The `varnishtest` tool performs Varnish unit tests based on *VTC files* containing unit testing scenarios
Advanced logging	The `varnishlog`, `varnishtop`, and `varnishncsa` tools allow you to perform deep introspection into the Varnish flow, the input and output
Advanced statistics	The `varnishstat` tool displays numerous counters that give you a global insight into the state of your Varnish server

Additionally, *Varnish Cache* also comes with a set of *VMODs* that are plugged into Varnish, and which are discussed in *chapter 5*.

1.3.4 Which features does Varnish Enterprise have?

Here's a list of some of the core features of *Varnish Enterprise*:

Feature	Description
Massive Storage Engine *(MSE)*	An optimized dual-layer storage solution that offers persistence
Varnish High Availability *(VHA)*	A multi-master object replication suite that keeps the contents of multiple Varnish servers in sync, and as a consequence reduces the number of backend revalidation requests
Varnish Controller	A GUI and API to administer all the Varnish servers in your setup
Varnish Custom Statistics *(VCS)*	A statistics engine allowing you to aggregate, display and analyze user web traffic and cache performance in real time
Varnish Broadcaster	Broadcasts client requests to multiple Varnish nodes from a single entry point
Varnish Live	A mobile app that shows the performance of Varnish instances
Varnish Web Application Firewall *(WAF)*	Web Application Firewall capabilities, based on the *ModSecurity* library
Client TLS/SS	Termination of client *TLS/SSL* connections *on the edge*
Backend TLS/SS	Connect to backend servers over *TLS/SSL*, ensuring *end-to-end encryption*
Parallel ESI	Process *Edge Side Includes (ESI)* in parallel, whereas *Varnish Cache* only processes *ESI tags* sequentially
JSON logging	Output from the `varnishlog` and `varnishncsa` logging tools can be sent in *JSON* format
TCP-only probes	Allow probes to perform health checks on backends by checking for an available TCP connection, without actually sending an HTTP request

`last_byte_timeout`	A backend configuration parameter that defines how long Varnish waits for the full backend response to be completed
Total Encryption	Encryption of cached objects, both in memory and on disk
Veribot	Identify and verify traffic that comes from online bots
Brotli compression	Compress HTTP responses with *Brotli* compression, which offers a higher compression rate than *GZIP*
Dynamic backends	Define backends *on-the-fly*, instead of relying on hardcoded backend definitions in the *VCL file*
Body access & modification	Via the *xbody* module, request and response bodies can be inspected and modified

Besides feature additions in the *Varnish core*, *Varnish Enterprise* offers many features as *VMODs* that are plugged into *Varnish Enterprise*, which are also discussed in *chapter 5*.

1.4 Which use cases does Varnish address?

Historically Varnish has always been associated with *web acceleration*. Varnish was invented to speed up websites and the majority of Varnish users have a *web acceleration use case*.

However, Varnish is not solely built for websites: Varnish is an *HTTP accelerator* and there are far more HTTP use cases than just websites.

1.4.1 API acceleration

Accelerating APIs is a good example of an alternate use case for Varnish: APIs return HTTP responses and interpret HTTP requests, but they do not return *HTML output*. In most cases a *REST API* will return *JSON* or *XML*.

One could say that the acceleration of *REST APIs* is more straightforward than speeding up a website, and that is because *REST APIs* inherently respect the best practices of HTTP caching:

- The *HTTP request method* indicates the type of action that is taken
- The *idempotency* of *HTTP request methods* is respected, and by design only *GET* and *HEAD* requests are cached
- *Cookies* are hardly ever used in a *RESTful context*, which makes caching a lot easier
- The hierarchical nature of URLs and how they represent their respective resource is very intuitive

API authentication is a more complicated matter: as soon as an `Authorization` header appears, caching usually goes out the window. Just like *cookies*, auth headers are a mechanism to keep track of *state*. They imply that the data is *for your eyes only* and hence cannot be cached.

Of course Varnish has an elegant way to work around these limitations, but we'll talk about *state* and *authentication on the edge* at a later stage in the book.

1.4.2 Web acceleration

Let’s rewind for a minute, and focus on website acceleration.

Generating the HTML markup for a dynamic website is often done using server-side programming languages like PHP, Python, ASP.NET, Ruby, or Node.js. These languages have the ability to interact with databases and APIs. Although they seem quite fast, they are prone to heavy load as soon as the concurrency increases.

The concurrency aspect is very significant: yes, the application logic that generates HTML output will consume CPU, RAM and disk I/O. But the most time is spent waiting for external resources, such as databases or APIs. While your application is waiting, resources cannot be freed and the connection between the client and the server remains open for the duration of the request.

On a small scale, this has no impact on the user experience, but at larger scale, more memory and CPU will be used, and a lot more time is spent waiting for results to be returned by databases. And eventually you’ll run out of available connections, you’ll run out of available memory, and your CPU usage may spike.

The stability of your entire website is in jeopardy. These are all problems that weren’t tangible at small scale, but the risk was always there.

The bottom line is that code has an impact on the performance of the server. This impact is amplified by the concurrency of visits to your website. By putting Varnish in front of your web server, the impact of the code is mitigated: by caching the HTML output in Varnish, user requests can immediately be satisfied without having to execute the application logic.

> The resource consumption and stability aspect also applies to APIs of course, and to any HTTP platform that requires a lot of computation to generate output.

1.4.3 Private CDN

You’ll also notice that websites consist of more than text formatted in HTML markup:

- There are lots of images
- *CSS files* are used to improve the look and feel
- *JavaScript files* are used to make websites more interactive
- Custom web fonts are loaded through *WOFF files*

All these files and documents need to be accelerated too. Some of them are a lot bigger in size. Although modern web servers don't need a lot of CPU power or memory to serve them, there are bottlenecks along the way that require a *reverse caching proxy* like Varnish.

As mentioned before: web servers only have a limited number of available connections. At large scale you quickly run out of available connections. Using Varnish will mitigate that risk.

Another aspect is the geographical distance between the user and the server, and the latency issues that come into play: transmitting images and other large files to the other side of the world will increase latency. That's just physics: light can only travel so fast through fiber-optic cables.

Having servers close to your users will reduce that latency, which has a positive impact on the quality of experience. By putting Varnish servers in different locations, you can efficiently reduce latency, but you also horizontally scale out the capacity of your web platform, both in terms of server load and bandwidth.

Let's talk about bandwidth for a minute. At scale, the first problem you'll encounter is a lack of server resources. With Varnish, you'll be able to handle a lot more concurrent users, which will expose the next hurdle: a lack of bandwidth.

Your web/HTTP platform might have limited network throughput. Your network may be throttled. Maybe you operate at such a scale that you don't have sufficient network resources at your disposal.

In those cases it also makes sense to distribute your Varnish servers across various locations: not just to reduce latency, but also to be on multiple networks that have the required capacity.

> This use case may sound familiar, and it is exactly the problem that a *content delivery network (CDN)* tries to tackle: by placing caching nodes in different *points of presence (PoPs)*, latency is reduced, network traffic to a single server is reduced, and excessive server load is tackled as well.

Varnish can serve as a *private content delivery network (Private CDN)*, accelerating content close to the consumer. Even caching large volumes of content is not a problem: setting up a *multi-tier* Varnish architecture with *edge nodes* for *hot content* and *storage nodes* to store more content sharded over multiple nodes allows you to cache petabytes of data using horizontally scalable architecture.

Varnish Enterprise even has a *purpose-built stevedore* that combines memory and optimized disk storage to build your own Private CDN. It's called the *Massive Storage Engine* and it is covered in depth in *chapter 7*.

The *why* and the *how* of *Private CDNs* is further explained in *chapter 9*.

1.4.4 Video streaming acceleration

A more unexpected use case for Varnish is the acceleration of *online video streaming platforms*, or *OTT platforms* as we call them. More than 80% of the internet's bandwidth is used to serve video. These are staggering numbers, and video has its own unique content delivery challenges.

Online video is not distributed using traditional broadcast networks, but *over the top (OTT)*. Meaning that a third-party network, in this case the internet, is used to deliver the content to viewers. The distribution of this type of video also uses *HTTP* as its go-to protocol. And once again, Varnish is the perfect fit to accelerate *OTT video*.

Accelerating video has many similarities with *Private CDN*:

- Online video consists of large volumes of data that need to be transferred over the internet.
- Encoding and packaging video into the right format is very resource intensive, and at scale this requires a lot of server capacity.
- Latency has a negative effect on the *quality of experience*.
- Putting cached video content closer to the viewers, and scaling out the delivery, will reduce latency and reduce the load on the origin.

Although it seems video streaming acceleration is a carbon copy of a regular *Private CDN*, there are some unique challenges.

A lot of it has to do with how online video is packaged. *OTT video*, both live and on demand, is chopped up into segments. Each segment represents on average six seconds of video. This means that a video player has to fetch the next segment every six seconds. For *4K* video, a six-second segment requires transmitting between 10 MB and 20 MB of data. Audio can be a separate download stream and this also applies to subtitles.

A single *4K* stream consumes at least 6 GB per hour. Not only does this pose an enormous bandwidth challenge, low latency is also important for the continuity of the video stream, and of course the quality of experience. The slightest delay would result in the video player having to rebuffer the content.

Some of Varnish's features are ideal for caching live video streams, guaranteeing low latency. For *video on demand (VoD)*, the enterprise product has the storage capabilities as well as a module to prefetch the next video segment.

Varnish for *OTT video* is discussed in depth in *chapter 10*.

1.4.5 Web application firewalling

Varnish operates *on the edge*, which is the outer tier of your web platform. It is responsible for handling requests from the outside world.

From an operational point of view, the *outside world* comes with a lot risk. Ensuring the stability of your platform is key, and a *reverse caching proxy* like Varnish has an important role in maintaining that stability.

We already talked about performance. We also talked about scalability, which is maintaining performance and stability at large scale. These are some of the risks that we try to mitigate.

Another important aspect of risk mitigation is security: it's not always a large number of concurrent visitors that jeopardizes stability; it's also about what these visitors do when they're on your platform.

Websites, APIs, content delivery solutions all consist of many pieces of software. Often all of this software has layer upon layer of components, third-party libraries, and tons of business logic that is written in-house. Keeping all that software secure is a massive undertaking. From the operating system to the web server, from encryption libraries to the component that allows your code to interact with the database: more than 90% of the code is written and maintained by third parties.

Although many organizations have the discipline of installing security updates as soon as they become available, it's not always clear what needs to be patched. Hackers and cybercriminals are a lot more aware of the vulnerabilities out there, and they're not afraid to exploit them.

There are many *VCL code snippets* available that try to detect malicious access to web platforms: from *SQL injections*, to *Cross Site Scripting* attempts. In this context, Varnish assumes the role of a *web application firewall (WAF)*, blocking malicious requests. Although these VCL code snippets work to some extent, they are hard to maintain, and are hardly as effective as well-respected *WAF* projects like *ModSecurity*.

Varnish Enterprise has a *WAF* add-on module that wraps around *ModSecurity*. It allows for all traffic to be inspected by ModSecurity and is configurable using VCL. Suspicious requests are blocked and never reach your origin.

The *Varnish WAF* supports all *ModSecurity* features and the full rule set, including the *OWASP Core Rule Set*. This includes:

- SQL Injection (SQLi)
- Cross Site Scripting (XSS)
- Local File Inclusion (LFI)
- Remote File Inclusion (RFI)
- Remote Code Execution (RCE)
- PHP Code Injection
- HTTP Protocol Violations
- HTTPoxy
- Shellshock
- Session Fixation
- Scanner Detection
- Metadata/Error Leakages
- Project Honey Pot Blacklist
- GeoIP Country Blocking

When a security vulnerability is detected and reported in the list of *common vulnerabilities and exposures (CVE)*, it is expected that the vulnerability is fixed at the source. Unfortunately software maintainers aren't always quick enough to respond in time. Luckily *ModSecurity* proactively releases new rules to protect your origin against so-called *zero-day attacks*.

The *Varnish WAF* and security in general will be covered in much more detail in *chapter 7*.

1.5 Under the hood

Now that you know what *Varnish* does, and how powerful this piece of software is, it's time to take a look behind the curtain. The raw power of *Varnish* is the direct consequence of an architecture that doesn't compromise when it comes to performance.

This section is very technical, and covers some concepts that will be clarified later in the book. However, it is useful to talk about some of the internals of *Varnish* right now, because it will give you a better understanding when we cover the concepts in details.

This applies to *runtime parameters*, default behavior, and the impact of certain *VCL* changes.

Let's start at the beginning.

When the `varnishd` program is executed, it starts one additional process resulting in:

- The manager process
- The child process

In `varnishd` there is separation of privileges, where actions that require *privileged access* to the operating system are run in the manager process. All other actions run in a separate child process.

1.5.1 The manager process

Because of its privileged access to the operating system, the manager process will only perform actions that require this access, and leave all other tasks to the child process.

The manager process is responsible for opening up sockets, and binding them to a selected endpoint.

If you configured `varnishd` to listen for incoming connections on *port 80*, which is a privileged port, this is processed by the manager process with elevated privileges. However, the manager process will not handle any data that is sent on this socket. *Listening* on that socket, *accepting* new connections, and reading incoming data is the responsibility of the child process.

The manager process will also make sure that the child process is responsive. It continuously pings the child process, and if the child process should become unresponsive or die unexpectedly, the manager process will tear down the old and start a new one.

The manager process also owns the *command line socket*, which is used by the `varnishadm` management program. Privileged access is required here in order to start or stop the child process using `varnishadm`.

The manager process is also responsible for opening up the *VCL file* and reading its contents. However, the compilation of *VCL* happens in a separate process.

And finally, the manager process is also responsible for the different *VCL configurations* that were loaded into Varnish.

1.5.2 The VCL compiler process

As mentioned, the *manager process* will open the *VCL file*, and will read the contents, but it will not process the *VCL code*. The *VCL compiler process*, which is a separate process, will take care of the *VCL compilation*.

The process is named `vcc-compiler`. However, you won't often see it appear in your process list, as it is a transient process: it only runs for the duration of the compilation.

The `vcc-compiler` process isn't just used on startup; it also runs when the `vcl.load` or `vcl.inline` commands are executed by the `varnishadm` *command line tool*.

Compilation steps

The first step in the *VCL compilation process* is to take the raw *VCL code* read from the *VCL file* and process any include statements in the VCL code. The files are resolved and inserted verbatim in the code. After that a special source referred to as the *built-in VCL code* is included last. This adds a sane default behavior that respects best practices of each *VCL function*, even if you don't write any additional *VCL*.

> As mentioned earlier, *VCL* is language can be used to extend the behavior of or various states in the *Varnish finite state machine*. The *built-in VCL* is just there as a safety net. The details of this *finite state machine* will be covered in *chapters 3 and 4*.

With the complete VCL source available, the VCL compiler will then transform the VCL code into *C-code*. The management process will then spawn a new `vcc-compiler` process to compile the *C-code*.

> If you want to see the actual *C-code* that is produced for your *VCL file*, you can run the following command: `varnishd -C -f <vcl_filename>`.

The `vcc-compiler` will take the *C-code*, and will run the `gcc` compiler to compile the code. This compiles and optimizes the *C statements* into object code that can be executed directly by the host system CPU. The output is a *shared library* under the form of a `.so` file. The *parameter* `cc_command` is used by the `vcc-compiler` to set the *gcc flags* used for compilation.

After that, the *child process* is messaged, and will use and run the `.so` file.

All these steps describe how *Varnish* compiles the *VCL* on startup. But using the `varnishadm vcl.load`, and the `varnishadm vcl.use` commands, you can load new *VCL* at runtime. `varnishadm vcl.load` will compile the *VCL* into the `.so` file and load it. `varnishadm vcl.use` will select the `.so` file to be used on new connections.

1.5.3 The child process

From a security point of view, you really want to avoid giving the manager process too many responsibilities. That's why the child process does most of the work in Varnish.

Accepting the connections, processing the requests, and producing the responses are all done in the child process. Seen from afar the child process basically sits in a loop and waits for incoming connections and requests to be processed. This in turn will activate the numerous mechanisms that make up the Varnish caching engine, such as backend fetches and caching of content.

1.5.4 Threads

To be honest, all this logic doesn't happen in one place. The child process will distribute the workload across a set of threads. Threads are used to facilitate both parallelism and asynchronous operations.

There are various threads in *Varnish*. Some of them have a dedicated role, others are general-purpose worker threads that are kept in thread pools.

Here's an overview of the *threading model in Varnish*:

Thread name	Amount	Task
`cache-main`	1	startup & initialization
`acceptor`	1 per thread pool per listening endpoint	accept new connections
`cache-worker`	1 per active connection	request handling, fetch processing and probe execution
`ban-lurker`	1	background ban processing and ban list cleaning
`waiter`	1	manages idle connections
`expiry`	1	remove expired content
`backend-poller`	1	manage probe tasks
`thread-pool-herder`	1 per thread pool	monitor & manage threads
`hcb_cleaner`	1	cleaning up retired hashes

If you're using *Varnish Enterprise*, you can use the `varnishscoreboard` program to display the state of the currently active threads.

Varnish Enterprise has several additional threads

Thread name	Amount	Task
`vsm_publish`	1	publish & remove shared memory segments
`cache-memory-stats`	1	memory statistics gathering
`cache-governator`	1	memory governor balancing thread
`mse_waterlevel`	1 per MSE book	MSE book database waterlevel handling
`mse_aio`	1 per MSE store	MSE store AIO execution
`mse_hoic`	1 per MSE store	MSE store waterlevel handling

The cache-main thread

The `cache-main` thread is the entry point at which the management process forks off the child process.

This thread initializes the dependencies of the child process. These are just a set of dedicated threads, which will be covered in a just a minute.

As soon as the initialization is finished, the `cache-main` thread really doesn't have anything more to do. So it turns into the *command line thread*: it sits in a loop, waiting for *CLI commands* to come in.

This may seem confusing because earlier I mentioned that the *management process* takes care of the command line. Well, in fact, they both do.

There is a *Unix pipe* in between management process and the `cache-main` thread of the *child process*. Although the *command line socket* is owned by the manager process, commands that are relevant to the child process, will be sent over the Unix pipe.

Commands that require privileged access to the system are the responsibility of the manager process.

The thread pool herder thread

One of the first threads that is initialized by `cache-main` is the `thread-pool-herder` thread because a lot of internal components depend on thread pools.

A thread pool is a collection of resources that *Varnish* reuses while handling incoming requests. These resources include things like the *worker threads* and the *workspaces* they use for *scratch space*. Some of these resources benefit from *Non-uniform memory access (NUMA) locality*, and are grouped together in a pool. The number of thread pools is configurable through the `thread_pools` runtime parameter. The default value is *two*.

> A thread pool manages a set of threads that perform work on demand. The threads do not terminate right away. When one of the threads completes a task, the thread becomes idle, is returned to the pool, and is ready to be dispatched to another task.

The `thread-pool-herder` is a per pool management thread. It will create the amount of threads that is defined by the `thread_pool_min` runtime parameter at startup, and never goes below that amount. The default value is *100*.

When the traffic on the *Varnish* server increases, the `thread-pool-herder` threads will create new threads in their pools. It will continue to monitor the traffic, and create new threads until the `thread_pool_max` value is reached. The default value is *5000*.

Note that `thread_pool_min` and `thread_pool_max` set limits per thread pool.

When new workload exceeds the amount of free threads, the `thread-pool-herder` thread will queue incoming tasks, while new threads are being created.

When threads have been idle for too long, the `thread-pool-herder` thread will remove these threads from the thread pool. The `thread_pool_timeout` runtime parameter defines the thread idle threshold.

But as mentioned, the number of threads in a thread pool will never go below the value of `thread_pool_min`.

The acceptor threads

The `acceptor` threads are the *point of entry* for incoming connections. They are created by the `cache-main` thread, and are one of those dependencies I referred to.

The `acceptor` threads will call `accept` on a socket that was opened by the management process. This call is the server end part of the *TCP handshake*. The `SYN-ACK` part, if you will.

The *acceptor threads* will then delegate the incoming connections by dispatching a *worker thread* from the thread pool.

There is *one acceptor thread per listening point per thread pool*. This means for a single listening point and the default number of thread pools, there will be two acceptor threads that are running.

The waiter thread

The `waiter` thread is used to *manage idle file descriptors*.

Behind the scenes, `epoll` or `kqueue` are used, depending on the operating system. `epoll` is a *Linux* implementation. `kqueue` is a *BSD* implementation. Since this book focuses on Linux, we'll talk about `epoll`.

`epoll` is the successor of the `poll` system call. It polls file descriptors to see if *I/O* is possible. `epoll` is a lot more efficient at large scale. The same applies to `kqueue` on *BSD* systems.

The term *file descriptor* is quite vague because we know that on Unix systems everything is a file. Network connections also use file descriptors, and *Varnish* happens to process a lot of those at large scale.

Varnish leverages the `waiter` to keep track of open backend connections. Whenever a backend connection is idle, it will sit in the waiter for *Varnish* to monitor the connection status.

In addition, *Varnish* will use the `waiter` for client connections whenever we are done processing a request and the connection goes idle.

Varnish does not use `epoll` for regular connection handling: client traffic is still processed using *blocking I/O*. `epoll` is only used for idle connections.

The expiry thread

The `expiry` thread is used to remove *expired objects* from cache.

This thread keeps a *heap data structure* that tracks the *TTL* of objects. The object that expires next is always at the top of the data structure.

When an expired object is removed, the *heap* is re-ordered and again has the object that expires next at the top.

The `expiry` thread removes expired objects, goes back to sleep, and wakes up to do it all over again. The amount of time that the `expiry` thread sleeps is the time until the new element at the top of the heap expires.

The backend-poller thread

The `backend-poller` thread manages a set of *health probe tasks*. Health probes are used to monitor the health of backends, and to decide whether or not a backend can be considered *healthy*.

The `backend-poller` thread keeps track of the *health check interval* that was defined by the probe, and dispatches the health check at the right time.

As mentioned, this thread *manages* probes, and *dispatches* health checks. It doesn't perform the actual *HTTP request* itself. Instead the `backend-poller` thread will farm out the work to a *worker thread*.

The ban-lurker thread

The `ban-lurker` thread has the responsibility of removing items from the *ban list*.

But before we can talk about that, let me briefly explain what *banning* means.

Banning, and content invalidation will be covered in detail in *chapter 6*.

A *ban* is a mechanism in *Varnish* to ultimately remove one or more objects from the cache.

Bans happen based on a *ban expression*, and these bans end up on the *ban list*. This expression can be triggered using the `ban()` function in *VCL*, or by calling the `ban` command in the `varnishadm` administration tool.

Expressions that match objects in the cache cause these objects to be removed from cache. Once all objects have been checked, the *ban* is removed from the *ban list*, because it is no longer relevant.

Bans are evaluated when an object is accessed, causing a ban expression to have an immediate effect on the cache. The `ban-lurker` thread is responsible for matching *ban expression on the ban list* with all the objects in cache, as well as those that are infrequently accessed.

There are some *runtime parameters* that influence the behavior of the thread:

- `ban_lurker_age`: the age a ban should have before the lurker evaluates it. The default value is *60 seconds*
- `ban_lurker_batch`: the number of bans that are processed during a ban lurker run. The default value is *1000*
- `ban_lurker_holdoff`: the number of seconds the ban lurker holds off when locking contention occurs. The default value is *0.010 seconds*
- `ban_lurker_sleep`: the number of seconds the `ban-lurker` thread sleeps before performing its next run. The default value is also *0.010 seconds*

Worker threads

There is one worker thread per active connection when the `HTTP/1` protocol is used. For `HTTP/2` there are multiple worker threads per connection: One for the `HTTP/2` session, and one for each `HTTP/2` *stream*.

Additionally, each backend fetch will consume one worker thread.

Worker threads can be spawned on demand, and the cost of spawning new threads comes at a cost. That's why we pre-allocated a number of threads in the thread pools.

1.5.5 Transports

One of the first tasks that the *worker thread* performs is checking which *protocol handler* was configured.

In *Varnish Cache*, this can be the *PROXY transport handler*, or the regular *HTTP transport handler*.

In *Varnish Enterprise*, there's the addition of the *TLS transport handler*.

Imagine the following *address configuration* in Varnish:

```
varnishd -a :80,PROXY -f /etc/varnish/default.vcl
```

Because `PROXY` was used, we first need to handle the *PROXY protocol* bytes that are part of the *TCP preamble*. This information contains the *IP and port* that were used to connect to *Varnish*.

Varnish will populate the various *IP and port variables* based on the information.

Once this decoding process is finished, the *PROXY transport handler* will hand off the work to the *HTTP transport handler*, which is now able to process the HTTP part of the *TCP request*.

The *HTTP transport handler* will parse the HTTP request and populate the necessary internal data structures with request information for later use.

In the example below, we're not using the *PROXY protocol*, this means the *HTTP transport handler* is used immediately:

```
varnishd -a :80 -f /etc/varnish/default.vcl
```

In *Varnish Enterprise*, we can configure *native TLS support*. Our *address configuration* may look like this:

```
varnishd -A /etc/varnish/tls.cfg -a :80 -f /etc/varnish/default.vcl
```

Because the `-A` runtime parameter was used, the *TLS transport handler* will be used, which will handle the *crypto part*. But after that, the work is handed off to the *HTTP transport handler*.

1.5.6 Disembarking

When a *worker thread* is waiting for a *fetch* to finish, its internal state can be stored on a waiting list, while the worker thread is put back into the thread pool.

This concept is called *disembarking*, and is an optimization so as to avoid needlessly tying up resources that are waiting.

Transactions on the waiting list can be woken up after a fetch finishes, and will be redispatched to another worker thread.

1.5.7 The waiting list

When an incoming request doesn't result in a cache hit, Varnish has to connect to the *origin server* to fetch the content. If a lot of connections for the same resource happen at the same time, the *origin server* has to process a lot of connections through Varnish, and could suffer from increased server load.

To avoid this, *Varnish* has a waiting list per object, where requests asking for the same object are grouped together.

The first request for this object will result in the creation of a *busy object*, which tells Varnish there is a fetch in progress. While the busy object is in place, all subsequent requests for this resource are put on the waiting list.

As soon as the response is ready for delivery, all items on the waiting list can be satisfied. However, the *rush exponent* will make sure the kernel doesn't choke on a sudden increase of activity.

The `rush_exponent` runtime parameter defines the amount of waiting list items that can be processed exponentially. Its default value is *3*. This means that the first run will satisfy three objects, the next run will satisfy nine objects, and the following one will satisfy 27 objects. This is a mitigation put in place to avoid the so-called *thundering herd problem*.

The exponential nature of this mechanism ensures a workload buildup that the kernel will be able to handle.

This concept of satisfying multiple items on the waiting list is called *request coalescing* because we're basically *coalescing* multiple similar requests into a single backend request.

1.5.8 Serialization

Request coalescing is a very powerful feature in Varnish. But when Varnish is not able to get a proper *TTL* for the object, the object is immediately expired.

The first transaction on the waiting list will be satisfied by the fetch, but since the object was immediately expired it cannot be used to satisfy the rest of the requests on the waiting list.

This means that the other waiting list items are kept there, and are processed serially. This side effect is what we call *serialization* because the waiting list is processed serially, instead of in parallel.

As you can imagine, serialization is very bad for performance, and for the quality of experience in general.

Imagine that you have a waiting list of *1000 items*, and a backend fetch takes *two seconds* to be completed. When *serialization* takes effect, the last transaction in the waiting list has to wait *2000 seconds* until completion.

The sole reason for *serialization* is *bad VCL configuration*. As a *Varnish operator*, you have the flexibility to override many aspects of the behavior of the cache. The *TTL* and the *cacheability* of fetched responses are part of that.

Non-cacheable responses are also cached in the so-called *hit-for-miss* or *hit-for-pass* cache. In essence, we're caching the decision not to cache and by default this happens for a duration of *120 seconds*.

Items on the *hit-for-miss* or *hit-for-pass* cache will bypass the waiting list to avoid serialization.

A common, but bad, practice is setting the *TTL* of an object to *zero* in *VCL*, when deciding not to cache. This expires the object immediately, and the waiting list no longer has the required information.

The way uncacheable content should be approached is by setting the object to *uncacheable* in *VCL*, and ensuring a proper *TTL*, which will be beneficial for transactions in the waiting list.

The *built-in VCL*, which is covered in *chapters 3 and 4*, has the necessary behavior in place to protect *VCL operators* from falling into this trap.

When writing *custom VCL*, please try to fall back on the *built-in VCL* as much as possible. As we will discover later in the book: *built-in VCL* behavior takes effect when *VCL subroutines* don't explicitly return an action.

1.5.9 Workspaces

The concept of *workspaces* in *Varnish* is an optimization to lessen the strain on the *system memory allocator*. Memory allocation is expensive, especially for short-lived allocations.

Varnish will allocate a chunk of memory for each transaction. A very simple allocator within *Varnish* can hand out memory from that chunk. We call this *workspace memory*.

Different parts of Varnish, use *workspace memory* in various ways:

Request handling happens using *workspace memory*, and is sized using the `workspace_client` runtime parameter. The default value is *64 KB*. This means that the *client-side processing* of each request, and the subsequent response, receives 64 KB of memory per request.

For transactions that involve a backend fetch, a separate piece of *workspace memory* is used: the `workspace_backend` runtime parameter defines how much memory per backend request can be used. By default this is also *64 KB*.

The `workspace_session` runtime parameter reflects how much memory from workspace can be consumed for storing *session data*. This is mainly information about *TCP connection addresses*, and other information that is kept for the entire duration of the session. The default value here is *0.5 KB*.

There is also a `workspace_thread` runtime parameter that defines how much *auxiliary workspace memory* will be assigned as thread local scratch space. This memory space is primarily used for storing *I/O-vectors* during delivery.

1.5.10 Backend fetches

Ever since *Varnish 4*, there has been a split between client-side logic and backend-side logic. Whereas there was only one thread for this in *Varnish 3*, it got split up into two separate threads from *Varnish 4*.

A major advantage is that *background fetches* are supported. This means that a client doesn't need to wait for the backend response to be returned. A *background fetch* takes place, and while that happens, a *stale object* can be served to the client.

As soon as the *background fetch* is finished, the object is updated, and subsequent requests receive the *fresh data*.

Streaming

Another advantage of the client and backend split is that it enables *streaming deliver*. When this is enabled, the body of a backend fetch may be delivered to clients as it is being received.

This of course has the side effect that fetch failures become visible to the clients. The *streaming delivery* can be turned off if this is not desired by setting the `beresp.do_stream` VCL variable to `false` in `vcl_backend_response`. This will cause the entire object to be received before it is delivered to any waiting clients.

Varnish Fetch and Delivery Processors

When Varnish fetches content from a backend, it flows through a set of filters called *Varnish Fetch Processors (VFP)*. These filters perform different tasks, like compressing the object using *GZIP* or parsing the content for *Edge Side Include (ESI)* instructions.

Similarly when delivering content to clients a set of filters called *Varnish Deliver Processors (VDP)* is used. These typically perform tasks like decompressing content if necessary, or stitching together `ESI` content.

1.6 Chapter summary

By now, you should have a solid idea of what Varnish is, and how it is an important tool in the modern-day web stack.

Delivering content at scale proves to be a lot more challenging than anticipated.

We cannot ignore the importance of speed, scalability, and stability of web platforms. Without a well-thought-out content delivery strategy, you'll suffer from a lot more downtime, or you'll have to spend a lot more money on sufficient infrastructure.

Varnish does more than just website and API acceleration: Varnish is content delivery software. With Varnish you can build your own *CDN* and tailor it to your exact needs.

Not only can this *CDN* cache images, scripts and documents of all kinds. It is exceptionally well-suited to accelerate *OTT video streaming platforms*. This is a significant use case, as more than 80% of the internet's bandwidth is used to stream video.

In the next chapter we will focus specifically on *Varnish version 6*, what has changed, how it is supported, and where to get it.

Chapter 2: Varnish 6

In this chapter we'll be focusing on the specifics of *Varnish version 6*: why it's important to use this version, what it offers in terms of functionality, how it is supported, and how you can get a hold of the software.

The entire book has a *Varnish 6* focus, but this chapter specifically justifies its usage and puts things in the right perspective for the chapters to come.

2.1 Why Varnish 6?

Varnish 6 is the latest major version of the Varnish project, and although it sounds new and shiny, the first release of Varnish 6 dates all the way back to *March 15th 2018*, when *Varnish Cache 6.0.0* was announced.

Later that year *Varnish Enterprise 6* was released to all customers as version *6.0.1r1*. Prior to this, some *limited availability* versions were produced and tested by select customers. With the release of *6.0.1r1*, all the features from the *4.1 version* of the enterprise product had been brought to the latest version, and even more features unique to *Varnish Enterprise 6* were introduced. These features will be discussed later in this chapter.

Right now, *Varnish Cache 6.6.0* is the latest version, which was released on *March 15th 2021*. Even though the recent release has a much higher version number, only *6.0* has the status as a *long-term support (LTS)* release.

Such an *LTS* release will receive bugfixes and support for an extended length of time: at least for half a year after the next *LTS* has been announced. Varnish Software maintains this release, and provides packages for free to the general public.

All versions of *Varnish Cache* prior to *6.0* are deemed *end-of-life*, while Varnish Software still supports older versions of *Varnish Enterprise*.

2.1.1 A lot of old content out there

Varnish 6 is already two years old. But there is still a lot of content available, written for versions that have reached their *end-of-life* date.

A lot of it is based on *Varnish version 4.1*. I even wrote a book about Varnish 4, which is still available. And although a lot of the concepts and *VCL* examples still hold up, it is important to educate people on the current state of the project.

The Varnish development team strives to be backwards compatible. This means that almost all VCL examples written for *Varnish 4.0 and 4.1* will work in exactly the same way in Varnish 6. However, there are often better ways of doing things in the newer versions, so not all of the advice is still current.

This book focuses on version 6, and many of the VCL examples in this book will not work with older versions of Varnish.

2.1.2 Varnish versions vs VCL syntax versions

We must however make a clear distinction between the version of Varnish itself, and the *VCL syntax versions*.

When *Varnish 4.0.0* was released in 2014, there were a lot of breaking changes to the *VCL syntax*. Most notably, backend and client request handling was separated, introducing several new states in the *Varnish finite state machine*.

As a mechanism for handling developments in the language itself, at the time and in the future, *Varnish Cache 4.0* introduced a new requirement: from that version on, every valid VCL file must start with a line defining the VCL version to use. For a long time, only `vcl 4.0;` was allowed.

Throughout the years, Varnish version numbers have increased, major releases like Varnish 5 and 6 happened, but the syntax remained compatible.

For a lot of people, VCL is the way they interface with Varnish. If the *VCL syntax* doesn't change, it seems as though the project doesn't evolve either.

In reality, a lot of change has happened, and a lot of change is still taking place right now. When *Varnish 6* was released, *VCL syntax 4.1* was introduced, while using the *4.0 syntax* is still allowed.

Support for *Unix domain sockets (UDS)* in Varnish was the feature that required your VCL file to start with `vcl 4.1;`.

It is important to understand that even though the VCL language was kept at a constant version for a long time, capabilities were added to it. In other words, it was kept *backwards compatible*, but not *forward compatible*. Most of these additions were in form of new *VCL variables* like, for example, `req.grace`, or `local.socket`.

2.1.2 Encouraging upgrades

The main problem with outdated Varnish content on the internet, and in professional literature, is that it doesn't represent the most efficient way to tackle certain issues that Varnish is suited for.

The lack of *modern Varnish content* on blogs, on *GitHub*, and on platforms like *Stack Overflow*, does not encourage users the install the latest major version.

When sharing new *VCL snippets*, we should make sure they start with `vcl 4.1;`. Because that's how we ensure people run them on a recent version of Varnish.

Encouraging upgrades from a feature perspective is one way to approach this challenge.

Another equally compelling argument is the fact that Varnish 6 is faster than previous versions, is more stable, less resource consuming, and more secure.

Varnish Cache 6 is an active project, *Varnish Enterprise 6* is an active product. Both will continue to receive bugfixes, security updates, and feature additions. As a Varnish evangelist, I strongly advise you to upgrade to this version in your current setup, or to install this version on new setups.

2.1.3 It's the way forward

A conservative approach and critical thinking are generally good qualities for any operations engineer to have. They're part of the risk mitigation mindset that gives organizations peace of mind.

Sticking with older versions that you know and trust is a common strategy: *if it ain't broken, don't fix it*. However, as another saying goes: *you have to get with the times*:

- Varnish 6 is the foundation of future development
- New features will not be backported to *pre-v6* versions
- Varnish 6 is the *LTS* version and will receive bugfixes and security updates
- Varnish 6 is faster and more stable
- New *VMODs*, either by Varnish Software, or the broader community, are unlikely to be compatible with older versions

Do yourself a favor, upgrade to Varnish 6, even if Varnish is not a cornerstone of your setup or platform.

2.2 What's new in Varnish 6?

Varnish 6 is not a snapshot, and it doesn't represent a single release: it's the major version that groups a set of releases.

These are the seven Varnish versions that have been released so far:

- Varnish 6.0 *(released March 15th 2018)*
- Varnish 6.1 *(released September 17th 2018)*
- Varnish 6.2 *(released March 15th 2019)*
- Varnish 6.3 *(released September 17th 2019)*
- Varnish 6.4 *(released March 16th 2020)*
- Varnish 6.5 *(released September 15th 2020)*
- Varnish 6.6 *(released March 15th 2021)*

Because Varnish is on a *six-month release schedule*, the *minor versions* go up quite quickly.

That doesn't mean every release adds a lot of functionality. They happen on a regular basis, and when there's something new, it's added. Sometimes you end up with *feature-heavy releases*, sometimes there isn't a lot to talk about.

There are also occasional *maintenance releases*. These aren't really scheduled: they just happen when they happen, and they increase the *patch version*, e.g. *Varnish 6.0.3*. These maintenance releases contain bugfixes, enhancements, and sometimes even security fixes.

But in order to answer the question, and to tell you what's new in *Varnish 6*, we need to look at the individual *minor versions*.

2.2.1 What's new in Varnish 6.0?

Varnish 6.0 isn't all that *feature heavy*. The new major version is primarily justified by the work that was done under the hood.

However, there are two features that are quite impactful:

- *Varnish 6.0* supports the use of *UNIX domain sockets (UDS)*.
- *HTTP/2* support that was an experimental feature of *Varnish 5* is now considered stable.

UNIX domain sockets (UDS)

Varnish is a proxy and a cache, but it is often more correct to think of it as a component in an *HTTP-based application* or *application stack*, where HTTP is the main communication protocol. Usually, the HTTP calls are made over *TCP/IP*, where the peer can be very close or somewhere across the globe.

But these services aren't always located on different servers, so there isn't always a need for a protocol that can route traffic over the network. *TCP/IP* is complex, and relatively speaking, there can be a significant overhead and latency when using it, even within the same server.

When the web server is hosted on the same physical machine, you can communicate over *UDS* instead of *TCP/IP*. The same applies when you terminate *TLS* on the same machine: the connection between the TLS terminator and Varnish can be made over *UDS*.

So instead of connecting to the web server using the following backend definition:

```
vcl 4.0;

backend default {
    .host = "localhost";
    .port = "8080";
}
```

you can remove *TCP/IP* from the equation and use the `.path` attribute to connect to the web server over *UDS*:

```
vcl 4.1;

backend default {
    .path = "/var/run/webserver.sock";
}
```

It's also possible to listen for incoming Varnish connections over *UDS* by configuring the `-a` runtime option accordingly:

```
varnishd -a /var/run/varnish/varnish.sock,PROXY,user=vcache,group=-
varnish,mode=660
```

UDS support introduced a couple of new *VCL variables*:

- `local.endpoint`: the address of the `-a` socket the session was accepted on
- `local.socket`: the name of the `-a` socket the session was accepted on

These variables and the new `.path` attribute aren't available in Varnish versions older than *6.0*, so that forced a new *VCL version*.

When using *UDS* features in VCL, you have to make sure the VCL file starts with `vcl 4.1;`.

HTTP/2 support considered stable

There is not a whole lot to say about *HTTP/2* support. Yes, it's stable now, and there are a few new *VCL variables* related to this feature:

- `req.proto`
- `bereq.proto`
- `beresp.proto`
- `resp.proto`

These variables all expose the *HTTP protocol version* that is used for the request or response. These are either `HTTP/1.1` or `HTTP/2.0`.

Using these variables in your VCL file also requires the file to start with `vcl 4.1;`.

At the moment, *Varnish* does not support *HTTP/2* on the backend side, but the variables `bereq.proto` and `beresp.proto` are reserved for future versions of *Varnish*, which might support more protocols than *HTTP/1.1* and *HTTP/1.0*.

Other features in Varnish 6.0

The *shard director* that is part of `vmod_directors` received lots of improvements.

`vmod_unix` was also added to retrieve the *group id (GID)*, the *user id (UID)*, the *group name*, and the *user name* of incoming connections over *UDS*.

`vmod_proxy` is a useful addition for users who terminate TLS and connect the *TLS proxy* to Varnish using the *PROXY protocol*. This VMOD has the ability to retrieve *TLS information* from the *PROXY protocol*. Most importantly whether or not the initial connection from the client was made using *TLS*.

Varnish 6.0 re-introduced a couple of removed *VCL variables*:

- `req.ttl`: upper limit on the object age for cache lookups to return hit
- `req.grace`: upper limit on the object grace

These variables are used to limit the time that objects are served from cache. Here's an example of how they can be used:

```
vcl 4.1;
import std;

sub vcl_recv {
    if (std.healthy(req.backend_hint)) {
        set req.ttl = 1m;
        set req.grace = 10s;
    }
}

sub vcl_backend_response {
    set beresp.ttl = 1h;
    set beresp.grace = 10m;
}
```

Even though the *time to live (TTL)* is explicitly set to one hour in `vcl_backend_response`, setting `req.ttl` and `req.grace` will limit the *effective* TTL and grace for the current transaction. In other words, when the backend is healthy, *Varnish* will only serve objects for *one minute* with a *ten-second grace*. If it turns out that the backend is not *healthy*, the original *TTL* and *grace* values will be used.

More information about *TTL* and *grace* can be found in the next chapter.

Another feature worth mentioning is `std.fnmatch`, which is part of the *standard VMOD (*`vmod_std`*)*. This function will perform *shell-style pattern matching* on a string, whereas *PCRE-style pattern matching* is otherwise used in Varnish.

2.2.2 What's new in Varnish 6.1?

Varnish 6.1 is a periodic release. It's one of those releases when the release date is due, but apart from bugfixes and enhancements, there's nothing much more to say.

If you really care about the internals, you can always have a look at the most recent version of the changelog.

2.2.3 What's new in Varnish 6.2?

Varnish 6.2 introduced two new variables that allow you to check if a request will bypass the regular caching flow, and immediately fetch the data from the backend. This happens when the corresponding backend response is marked as *hit-for-miss* or *hit-for-pass*. The respective variables are `req.is_hitmiss` and `req.is_hitpass`.

There was also an adjustment in the flow of Varnish's *finite state machine*, which involved `return(miss)` being removed from `vcl_hit{}`. There are other ways to still trigger a *miss*, even when a *hit* took place, but it is no longer deemed a common scenario.

A couple of new *type conversion* functions were added to `vmod_std`.

A new `lookup` function was added to `vmod_directors` to look up individual backends by name.

The `varnishadm` command line tool can now return output in *JSON* format thanks to the `-j` option.

And logging programs like `varnishlog` and `varnishncsa` will perform better due to internal enhancements and the introduction of the `-R` rate-limiting option.

And as always: this version of Varnish features a number of bugfixes and enhancements.

2.2.4 What's new in Varnish 6.3?

Explicitly trigger `vcl_backend_error`

Varnish 6.3 allows you to explicitly return an error from `vcl_backend_fetch` and `vcl_backend_response`. This triggers the `vcl_backend_error` state and can be achieved using either of the statements below:

```
return (error);
return (error(503));
return (error(503, "Service Unavailable"));
```

This is the erroneous equivalent of `return (synth(200,"OK"));`

VMOD import changes

In *Varnish 6.3* it is now possible to import a *VMOD* multiple times. It's also possible to run them under a different name, as illustrated below:

```
vcl 4.1;
import directors as dir;

sub vcl_init {
    new rr = dir.round_robin();
}
```

Behavior change in `auto` VCL temperature state

When loading multiple *VCL files* using `varnishadm`, the behavior of the `auto` state with regard to the *VCL temperature* has changed: in previous releases, the *VCLs* could cool down and would remain cold. As of *Varnish 6.3* it works in both directions, and these *VCLs* can automatically warm up again when required.

`std.ip()` accepts optional port argument

The `std.ip()` function now accepts an optional *port* argument that overrides the default port value when it is called through `std.port()`. The default value is `80`.

This is the new definition of `std.ip()`:

```
IP ip(STRING s, [IP fallback], BOOL resolve=1, [STRING p])
```

Querying changes in VSL tools

VSL tools like `varnishlog` and `varnishncsa` have the ability to filter output based on *VSL queries* using the `-q` option.

Prior to *Varnish 6.3*, a new line in a `-q` statement was merely used as a token separator. As of *Varnish 6.3*, every new line in a `-q` statement is treated as a new query. Unless specified otherwise, the statements are joined with the *or operator*.

Here's an example:

```
varnishlog -q "
    BerespStatus < 400
    ReqUrl eq '/'
"
```

Prior to *Varnish 6.3*, the use of multiple `-q` options would result in the last query being selected. As of *Varnish 6.3*, multiple queries will be joined with the *or operator*, as illustrated below:

```
varnishlog -q "BerespStatus < 400" -q "ReqUrl eq '/'"
```

The uppercase `-Q` is now available and reads *stored VSL queries* from a file. The `-Q` option can also be used multiple times, which just adds queries with the *or operator* to any query specified by either `-q` or `-Q`.

Here's an example:

```
varnishlog -Q query1.vsl -Q query2.vsl
```

And finally the `varnishncsa` program can take an `-E` option, which includes *ESI transactions* in the output.

2.2.5 What's new in Varnish 6.4?

`if-range` support

Varnish 6.4 features support for the `if-range` request header. The value of that header is either a date or an ETag value.

The result is that *range requests* are only performed if the value of the `if-range` header either matches the `Last-Modified` date of the response, or if it matches the value of the `ETag` response header.

Long story short: the `if-range` header performs *conditional range requests*.

Import *`vmod_cookie`* from varnish_modules

Prior to *Varnish 6.4*, `vmod_cookie` had to be installed from the varnish_modules repository. You either had to compile it manually, or get it from the package manager of your operating system.

As of *Varnish 6.4*, `vmod_cookie` has been imported into the main tree and is now available by default.

vmod_cookie makes parsing, fetching, and modifying cookies a lot easier. Here's an example:

```
vcl 4.1;
import cookie;

sub vcl_recv {
    if (req.http.cookie) {
        cookie.parse(req.http.cookie);
        # Either delete the ones you want to get rid of:
        cookie.delete("cookie2");
        # or delete all but a few:
        cookie.keep("SESSIONID,PHPSESSID");

        # Store it back into req so it will be passed to the backend.
        set req.http.cookie = cookie.get_string();

        # If empty, unset so the builtin VCL can consider it for
caching.
        if (req.http.cookie == "") {
            unset req.http.cookie;
        }
    }
}
```

In *Varnish Enterprise*, vmod_cookieplus solves many of the same use cases in a similar way. The feature set of vmod_cookieplus is also a bit broader.

Defining *none* backends

In *Varnish 6.4*, it is possible to explicitly state that you do not want a default backend by writing `backend default none;`.

The symbol `none` is similar to `NULL`, `null` and `nullptr` in other languages and represents the absence of an actual backend.

Any attempt to use the `none` backend, which is not a backend, will result in a failure.

In some cases, specifying `none` as the default makes a lot of sense: for example, you might use *Varnish* to just generate *synthetic HTTP responses*.

Another use case is when you are using labels to split *VCL logic* into more than one *VCL*, and the *main VCL* simply jumps to other loaded *VCLs* through `return (vcl(...))`. In these cases, it does not make sense to define an actual backend in the *main VCL*.

Finally, `vmod_goto`, a *Varnish Enterprise VMOD*, provides dynamic backends. By using `goto.dns_director()` or `goto.dns_backend()`, `none` makes a good choice as the default backend.

Other VCL changes

There are also some smaller *VCL* changes in 6.4:

- `std.rollback(header)` can be used to roll back the value of that header to its original state.
- Numerical expressions can now be negative or negated as illustrated in the following hypothetical example: `set resp.http.ok = -std.integer("-200");`.
- The `+=` operator can be used to append data to headers and response bodies.

2.2.6 What's new in Varnish 6.5?

Strict CIDR checks on ACLs

Access control lists in *Varnish* support hostnames, individual IP addresses, and also subnets. These subnets use the *CIDR notation*.

An example is `192.168.0.0/24`. As of *Varnish Cache 6.5*, the use of a non-zero host part will result in an `Address/Netmask mismatch` warning during *VCL compilation*.

Here's an example of a such a mismatch:

```
acl myAcl {
    "192.168.0.10"/24;
}
```

The correct notation that is enforced as of *Varnish Cache 6.5*, is the following:

```
acl myAcl {
    "192.168.0.0"/24;
}
```

vcc_acl_pedantic parameter

The `vcc_acl_pedantic` runtime parameter can turn *ACL CIDR mismatch warnings* into actual errors when enabled. As mentioned, these errors are triggered when the host bits of a *CIDR* in an *ACL* aren't all-zero.

obj.can_esi

Varnish Cache 6.5 introduced the `obj.can_esi` variable, which returns a boolean. If the response that is stored in the object can be processed using *ESI*, it returns true.

A new .resolve() method

As of *Varnish Cache 6.5*, there is a new `.resolve()` method for backends and directors, which immediately *resolves* a backend. Explaining exactly what this means requires an understanding of what a *director* is. In short, a *director* organizes a set of backends, for example for load balancing, and selects one from the set when a backend is needed to carry out a backend request. The new `.resolve()` method forces the director to immediately select a concrete backend.

Note that this method is not related to *DNS resolution* when a backend is defined using a domain name. In these cases, the DNS lookup happens once, during compilation, and the IP address is constant throughout the lifetime of the VCL.

Read more about different types of directors in *chapter 5*.

Closing the connection

Varnish Cache 6.5 now allows you to explicitly close the connection in *VCL*, bypassing any potential *keep-alive* settings that were in place.

Here's an example of how this can be accomplished in *VCL*:

```
sub vcl_backend_response {
    if (beresp.backend == faulty_backend) {
        if (beresp.http.Connection) {
            set beresp.http.Connection += ", close";
        } else {
            set beresp.http.Connection = "close";
        }
    }
}
```

BLOB literal syntax

Binary large objects or *BLOBs* as we call them, now have a new literal syntax as of *Varnish Cache 6.5*. The format is `:<base64>:`.

Here's an example of such a literal:

```
:3k0f0yRKtKt7akzkyNsTGSDOJAZOQowTwKWhu5+kIu0=:
```

std.blobread()

Similar to `std.fileread()`, the new `std.blobread()` function will read data from disk, but it will return a *BLOB* rather than a *string*. This is ideal for reading binary files.

No connection is made to a backend administratively set as unhealthy

When a backend is explicitly set to unhealthy using `varnishadm backend.set_health`, *Varnish* will no longer attempt to connect to the backend. The *unhealthy* status is immediately noticed, and a *HTTP 503* error is returned.

Help screen in varnishstat

If you type `h` in `varnishstat`, you'll get a help page as of *Varnish Cache 6.5*, explaining the various controls.

2.2.7 What's new in Varnish 6.6?

Start Varnish without a backend

The `-b` and `-f` runtime parameters are mutually exclusive. Prior to *Varnish Cache 6.5* it was not possible to start *Varnish* without defining the `none` backend in your *VCL file*, which required using the `-f` option.

In *Varnish Cache 6.5* it is now possible to use `-b none` to start the `varnishd` program without having to use the `-f` parameter.

Header validation

Headers can now be validated against the rules set by *RFC7230*. By default header validation doesn't take place.

By adding the `-p feature=+validate_headers` runtime parameter to `varnishd`, header validation is enabled.

Vary notices

In *Varnish Cache 6.6* the number of cache variations is now limited by the newly introduced `vary_notice` parameter. The default value is ten.

When the number of cache variations exceeds this value, a `Notice` record will be added to the *Varnish Shared Memory Log (VSL)*.

Checking ban errors

The `ban()` function is now deprecated because it lacks the ability to evaluate its success. In *Varnish Cache 6.6* two new ban functions were added:

- `std.ban()`: performs the ban and either returns `true` or `false` depending on its success.
- `std.ban_error()`: if an error occurred in `std.ban()`, `std.ban_error()` will display a detailed error message.

Here's a tiny *VCL snippet* that illustrates its use:

```
if (std.ban(...)) {
  return(synth(200, "Ban added"));
} else {
  return(synth(400, std.ban_error()));
}
```

Modulus operator

Varnish Cache 6.6 added the modulus operator to *VCL*. This corresponds to `%` and can be used as follows:

```
vcl 4.1;

import std;

sub vcl_recv {
    set req.http.number = "4";
    if(std.integer(req.http.number) % 2 == 0) {
        return(synth(200,"Even"));
    } else {
        return(synth(200,"Odd"));
    }
}
```

New notation for long strings

In addition to the `{" ... "}` format for denoting long strings, *Varnish Cache 6.6* has now added the `""" ... """` format.

New built-in VCL

The *built-in VCL* has been reworked: VCL code has been split into small subroutines, to which custom VCL can prepend custom code.

VCL variable changes

The `client.identity` variable is now accessible on the backend side. Prior to the release of *Varnish Cache 6.6*, this variable was only accessible on the client side.

The variables `bereq.is_hitpass` and `bereq.is_hitmiss` have been added to the backend side. They match the corresponding `req.is_hitpass` and `req.is_hitmiss` that operate on the client side.

The `bereq.xid` variable is now also available in the `vcl_pipe` subroutine.

The `resp.proto` variable is now a read-only variable.

2.2.8 Backports to 6.0 LTS

Several of the features described above have been ported to the *6.0 long-term support release series of Varnish Cache*, and to the corresponding *Varnish Enterprise* offering. The changelog for *Varnish Cache 6.0 LTS* releases can be found via https://varnish-cache.org/, while *Varnish Enterprise* changes are available at https://docs.varnish-software.com/varnish-cache-plus/changelog/changes/.

2.3 Varnish Enterprise 6

2.3.1 The origin story

The release of *Varnish Enterprise 6* was a new starting point for Varnish Software's commercial version. In fact, it was so significant that it warranted a rebrand from *Varnish Cache Plus*, to *Varnish Enterprise*.

The previous version, which was called *Varnish Cache Plus 4.1*, was based on *Varnish Cache 4.1*. Although there were plenty of *Varnish Cache 5.x* releases, there was no commercial equivalent.

Varnish Software's goal is to release a stable version that can be supported for a long time. The cost of stabilizing the code, and keeping the version up to date was not worth it for a potential *Varnish Enterprise 5* release, given the high quality of *Varnish Cache Plus 4.1* at the time, and the lack of *killer features* in *Varnish Cache 5*. This changed with *Varnish Cache 6* when *HTTP/2* support was becoming quite stable, and *Unix domain sockets* provided a considerable performance gain.

Because *Varnish Software* is an important contributor to the *Varnish Cache* project, preparations were made long before the actual release of *Varnish Cache 6*. As a matter of fact, *Varnish Enterprise 6.0.0r0* was ready before *Varnish Cache 6.0.0* was released. However, it is important to mention that *Varnish Enterprise 6.0.0r0* was never released to the public. The first public version was *Varnish Enterprise 6.0.1r1* on September 20th 2018.

2.3.3 New features in Varnish Enterprise 6

Now that *Varnish Cache Plus* had turned into *Varnish Enterprise*, a couple of major features had to be developed and released on top of this milestone version.

The big one was *Massive Storage Engine (MSE) version 3*. The first version of *MSE* was introduced in the days of *Varnish Cache Plus 4.0*. The second incarnation of *MSE* debuted with the first public release of *Varnish Cache Plus 4.1*. With *Varnish Enterprise 6*, *MSE* has reached its third iteration, and each new version has improved on the previous ones.

> *Chapter 7* has a section dedicated to *MSE*, in which its architecture, configuration, and usage is explained in-depth.

Along with *MSE* came the release of *Ykey*, a *VMOD* that is used for *tag-based invalidation* of objects in the cache. It is the successor of *Xkey*, and was specifically developed to work well with *MSE*.

Another important new feature that was launched was *Total Encryption*: an *end-to-end encryption* feature that was written in VCL and leveraged the brand-new *vmod_crypto*.

But as mentioned before: since *Varnish Cache 6* is not a single release, or snapshot, *Varnish Enterprise 6* isn't either. With every release, new features were added.

Here's a quick overview of some *Varnish Enterprise 6* feature additions:

- *Varnish Enterprise 6.0.0r0 (unreleased)*: Varnish Total Encryption and `vmod_crypto`
- *Varnish Enterprise 6.0.0r1 (unreleased)*: `vmod_urlplus`
- *Varnish Enterprise 6.0.1r1*: the return of `req.grace`
- *Varnish Enterprise 6.0.1r3*: `vmod_synthbackend`, MSE3
- *Varnish Enterprise 6.0.2r1*: `vmod_ykey`
- *Varnish Enterprise 6.0.3r6*: Varnish High Availability 6
- *Varnish Enterprise 6.0.3r7*: `vmod_mmdb` and `vmod_utils`
- *Varnish Enterprise 6.0.4r1*: `return(error())` syntax in `vcl_backend_fetch` and `vcl_backend_response`
- *Varnish Enterprise 6.0.4r2*: JSON formatting support in `varnishncsa`
- *Varnish Enterprise 6.0.4r3*: `vmod_str`
- *Varnish Enterprise 6.0.5r1*: `vmod_mse`, `last_byte_timeout` support for fetches
- *Varnish Enterprise 6.0.5r2*: `if-range` support in conditional fetches
- *Varnish Enterprise 6.0.6r2*: built-in TLS support, memory governor, `vmod_jwt`
- *Varnish Enterprise 6.0.6r3*: `vmod_stale`, `vmod_sqlite3`
- *Varnish Enterprise 6.0.6r5*: `vmod_tls`
- *Varnish Enterprise 6.0.6r6*: `vmod_headerplus`
- *Varnish Enterprise 6.0.6r8*: `vmod_resolver` and *Veribot*
- *Varnish Enterprise 6.0.6r10*: `vmod_brotli`
- *Varnish Enterprise 6.0.7r1*: `vmod_format`, a brand-new `varnishscoreboard`
- *Varnish Enterprise 6.0.7r2*: new counters for `vmod_stale`

- *Varnish Enterprise 6.0.7r3*: a new `resp.send_timeout` variable, `varnishncsa` EPOCH support, and the introduction of `utils.waitinglist()` and `utils.backend_misses()`
- *Varnish Enterprise 6.0.8r1*: `std.bytes()` was backported from *Varnish Cache*, introduction of `utils.hash_ignore_vary()`

It's important to know that the above feature list only covers the introduction of new features. Every release since *Varnish Enterprise 6.0.0r0* has seen improvements and added functionality to one more *Varnish Enterprise 6* features.

A feature is never really *done*, as we'll always be able to improve as we go.

So let's talk about individual features and show some VCL code.

Total encryption and vmod_crypto

At its base `vmod_crypto` consists of a set of cryptographic functions that perform various tasks.

Encoding

One example is the `crypto.hex_encode()` function that turns a *blob* into its hexadecimal value.

The VCL snippet below returns the hexadecimal value for `a`. Because `crypto.hex_encode()` accepts a *blob* as an argument, the `crypto.blob()` function is used to convert the *string* into the required *blob*.

```
vcl 4.1;

import crypto;

sub vcl_recv {
    return(synth(200, crypto.hex_encode(crypto.blob("a"))));
}
```

As you'd expect, the output returned by this VCL snippet is `61`. And using `crypto.string(crypto.hex_decode("61"))`, you can turn the hexadecimal value `61` back into `a`.

Similarly, we can also *base64 encode and decode*. Here's the *base64* equivalent:

```
vcl 4.1;

import crypto;

sub vcl_recv {
    return(synth(200, crypto.base64_encode(crypto.blob("a"))));
}
```

The output will be `YQ==`.

Hashing

Besides encoding, there are also hashing functions. The following example will create a *sha512* hash of a string:

```
vcl 4.1;

import crypto;

sub vcl_recv {
    return(synth(200, crypto.hex_encode(crypto.hash(sha512,"pass-
word"))));
}
```

`vmod_crypto` also supports *hash-based message authentication code* or *HMAC* as we call it.

The following example will create an *HMAC signature* for the string `password` using the `sha512` hash function, and will sign it using the secret key `abc123`:

```
vcl 4.1;

import crypto;

sub vcl_recv {
    return(synth(200, crypto.hex_encode(crypto.hmac(sha512,crypto.
blob("abc123"),"password"))));
}
```

The output will be `3e714097c7512f54901239ceceeb8596d2ced28e3b428ed0f82662c69664c11cc483daf01f66671fb9a7a2dac47977f12095dc08e1b2954e698de2220f83b97e`.

Encryption

And finally, `vmod_crypto` also supports encryption and decryption. The following example will return an *AES encrypted* string using a 16-byte key:

```
vcl 4.1;

import crypto;

sub vcl_recv {
    crypto.aes_key(crypto.blob("my-16-byte-value"));
    return(synth(200, crypto.hex_encode(crypto.aes_encrypt("pass-
word"))));
}
```

The output will be `60ed8326cfb1ec02359fff4a73fe7e0c`. And by calling `crypto.aes_decrypt(crypto.hex_decode("60ed8326cfb1ec02359fff4a73fe7e0c"))`, the encrypted value will be decrypted back to `password`.

Total Encryption

These cryptographic functions can be used in your VCL code, but they are also leveraged by *Total Encryption* to provide a service that encrypts your data before it gets stored in cache.

Encrypting your cached data is quite easy, and all the logic is hidden behind this one *include statement*:

```
include "total-encryption/random_key.vcl";
```

This statement will encrypt the response body using `crypto.aes_encrypt_response()` before storing it in cache. On the way out, it will decrypt the cached response body using `crypto.aes_decrypt_response()` before sending it to the client.

This approach uses a random key for encryption, where the key is stored in an *in-memory key-value store*. This is of course not persistent. That isn't really a problem because the cache itself is, by default, not persistent either.

However, if you use *disk persistence* in *MSE*, the key will not survive a restart, while the encrypted content will. We need a more reliable solution for that.

We can create a *secret key* on disk that is loaded into Varnish using the `-E` command line option. But first, we need to create the secret key. Here's a set of Linux commands to achieve this:

```
$ cat /dev/urandom | head -c 1024 > /etc/varnish/disk_secret
$ sudo chmod 600 /etc/varnish/disk_secret
$ sudo chown root: /etc/varnish/disk_secret
```

In order to set the `-E` option, you'll need to edit your *systemd unit file* and add the option. Here's an oversimplified example of how to do this:

```
ExecStart=/usr/sbin/varnishd ... -E /etc/varnish/disk_secret
```

But in the end, the only thing you'll need to add to your VCL file to support persisted data encryption is the following line:

```
include "total-encryption/secret_key.vcl";
```

vmod_urlplus

vmod_urlplus is a URL normalization, parsing and manipulation VMOD. It doesn't just handle the URL path, but also the query string parameters. It has a set of utility functions that make interacting with the URL quite easy.

The following example features a couple of *getter* functions that retrieve the *file extension* and *filename* of a URL:

```
vcl 4.1;

import urlplus;

sub vcl_backend_response {
    if (urlplus.get_extension() ~ "gif|jpg|jpeg|bmp|png|tiff|tif|img")
{
        set beresp.ttl = 1d;
    }
    if (urlplus.get_basename() == "favicon.ico") {
        set beresp.ttl = 1w;
    }
}
```

The next example is a normalization example in which we remove all query string parameters that start with `utm_`. Upon rewriting, the query string parameters are sorted alphabetically:

```
vcl 4.1;

import urlplus;

sub vcl_recv {
    # Remove all Google Analytics
    urlplus.query_delete_regex("utm_");

    # Sort query string and write URL out to req.url
    urlplus.write();
}
```

Normalizing URLs is useful when query string parameters are added that don't result in different content. Because items in cache are identified by the URL, removing these query string parameters results in fewer unnecessary cache variations, which increases your hit rate.

The example above explicitly removes query string parameters. Instead of stating which parameters should be removed, we can also state which ones we want to keep. The following example illustrates this feature:

```
vcl 4.1;

import urlplus;

sub vcl_recv {
    # Only keep id query string parameter
    urlplus.query_keep("id");

    # Sort query string and write URL out to req.url
    urlplus.write();
}
```

This example will remove all query string parameters, except the ones that were *kept*. In this case, only `id` will be kept.

The return of `req.grace`

`req.grace` is a VCL request variable that specifies the upper limit on the object grace.

Grace mode is a concept where expired objects are served from cache for a certain amount of time while a new version of the object is asynchronously fetched from the backend.

This feature was removed from Varnish in the *4.0.0* release, back in 2014, and was reintroduced in the *6.0.1* release in 2018.

Here's how to use it in your VCL code:

```
vcl 4.1;

import urlplus;
import std;

sub vcl_recv {
    if (std.healthy(req.backend_hint)) {
        set req.grace = 10s;
    }
}

sub vcl_backend_response {
    set beresp.grace = 1h;
}
```

So in this case we're setting the object's grace to an hour via `set beresp.grace = 1h;`, but as long as the backend is healthy, we're only allowing ten seconds of grace, via `set req.grace = 10s;`.

vmod_synthbackend

Varnish can return *synthetic responses*. These are HTTP responses that didn't originate from an actual backend request. The standard `return(synth(200,"OK"));` VCL implementation does a pretty decent job at this. Unfortunately, these responses are generated *on-the-fly*, and aren't cacheable.

`vmod_synthbackend` is a module that creates cacheable synthetic responses. Here's the VCL code for it:

```
vcl 4.1;

import synthbackend;

backend default none;

sub vcl_backend_fetch {
    set bereq.backend = synthbackend.from_string("URL: " + bereq.url
+ ", time: " + now);
}
```

The above example will print the current URL and the current timestamp. The response will be stored in cache using the default *TTL*.

Because backend requests and responses are *synthetic*, there is no need to define a real backend. Instead you can use `backend default none;` to set a *pro forma* backend called *default*.

MSE3

Version 3 of the *Massive Storage Engine* adds a lot more reliability and flexibility to the product. *Chapter 7* has a section dedicated to *MSE*, so I won't cover too many details here.

I'll just throw in an example configuration file that shows the capabilities of *MSE*:

```
env: {
    id = "myenv";
    memcache_size = "100GB";

    books = ( {
        id = "book1";
        directory = "/var/lib/mse/book1";
        database_size = "1G";

        stores = ( {
            id = "store-1-1";
            filename = "/var/lib/mse/stores/disk1/store-1-1.dat";
            size = "1T";
        }, {
            id = "store-1-2";
            filename = "/var/lib/mse/stores/disk2/store-1-2.dat";
            size = "1T";
        } );
    }, {
        id = "book2";
        directory = "/var/lib/mse/book2";
        database_size = "1G";

        stores = ( {
            id = "store-2-1";
            filename = "/var/lib/mse/stores/disk3/store-2-1.dat";
            size = "1T";
        }, {
            id = "store-2-2";
            filename = "/var/lib/mse/stores/disk4/store-2-2.dat";
            size = "1T";
        } );
    } );
};
```

Here's what this configuration file defines:

- My environment is called `myenv`
- The environment has 100 GB of memory to store objects
- The environment has two *books*, which are databases where metadata is stored
- Both books are 1 GB in size and are stored in `/var/lib/mse/book1` and `/var/lib/mse/book2`
- Each book has two *stores*, these are pre-allocated large files where objects are persisted
- Each store is 1 TB in size

- The stores for *book 1* are stored in `/var/lib/mse/stores/disk1/store-1-1.dat` and `/var/lib/mse/stores/disk2/store-1-2.dat`
- The stores for *book 2* are stored in `/var/lib/mse/stores/disk3/store-2-1.dat` and `/var/lib/mse/stores/disk4/store-2-2.dat`

There's a total of 4 TB for object storage, and 2 GB for metadata. Book and store allocation happens on a *round-robin basis*.

A specialized `mkfs.mse` program, which is shipped with your *Varnish Enterprise 6* installation, can be used to initialize all the files from the configuration file. This can be done as follows:

```
mkfs.mse -c /etc/varnish/mse.conf
```

Once these files have been initialized, it's a matter of linking the configuration file to Varnish using the `-s` option, as illustrated below:

```
ExecStart=/usr/sbin/varnishd ... -s mse,/etc/varnish/mse.conf
```

This is another relatively simple example. Because we will be going into much more detail about *MSE* in *chapter 7*, it suffices to understand that *MSE* is a very powerful stevedore that stores objects and metadata.

It combines the raw speed of memory with the reliability of persistent disk storage. It's truly a *best of both worlds* implementation that overcomes the typical limitations and performance penalties of disk storage.

vmod_ykey

As described earlier, `vmod_ykey` is the successor to `vmod_xkey`. It adds secondary keys to objects, which allows us to invalidate objects based on *tags*, rather than relying on the URL.

For implementations where content appears on many different URLs, it's sometimes hard to keep track of the URLs that need to be invalidated. `vmod_ykey` allows you to add tags to objects, and then invalidate objects based on those tags.

vmod_ykey will be covered more extensively in *chapter 6*. So let's keep it simple for now, and throw in one short VCL example:

```
vcl 4.1;

import ykey;

acl purgers { "127.0.0.1"; }

sub vcl_recv {
    if (req.method == "PURGE") {
        if (client.ip !~ purgers) {
            return (synth(403, "Forbidden"));
        }
        set req.http.n-gone = ykey.purge_header(req.http.Ykey-Purge,
sep=" ");
        return (synth(200, "Invalidated "+req.http.n-gone+" ob-
jects"));
    }
}

sub vcl_backend_response {
    ykey.add_header(beresp.http.Ykey);
    if (bereq.url ~ "^/content/image/") {
        ykey.add_key("image");
    }
}
```

This example will add the image tag to all objects that match URLs starting with /content/image/. The origin server, or application, can also issue tags via the custom Ykey response header. This is processed via ykey.add_header(beresp.http.Ykey);.

The logic in vcl_recv will process the invalidation by accepting HTTP requests using the PURGE request method, based on an *access control list*. The value of the custom Ykey-Purge header, is the tag that will be invalidated.

The HTTP request below can be used to invalidate all objects that match the image tag:

```
PURGE / HTTP/1.1
Ykey-Purge: image
```

Varnish High Availability 6

A new and improved version of *Varnish High Availability* was developed for *Varnish Enterprise 6*. Instead of relying on a dedicated agent for cache replication, *VHA6* leverages the *Varnish Broadcaster*, which is an existing component of the *Varnish Enterprise* stack.

By eating our own proverbial dog food, the implementation of *VHA6* is a lot simpler, and is written in VCL.

Chapter 7 will feature high availability in depth. For the sake of simplicity, a single VCL example will do for now.

Here's how you enable *Varnish High Availability* in *Varnish Enterprise 6*:

```
include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    call vha6_token_init;
}
```

The complexity is hidden behind the included file. And although there are plenty of configurable options, the secret token is the only value that needs to be set. If the Broadcaster's `nodes.conf` node inventory file is properly configured, *VHA6* will automatically synchronize newly stored objects with other Varnish nodes in the *cluster*.

vmod_mmdb

`vmod_mmdb` is the successor to `vmod_geoip`. Both *VMODs* are able to map a geographical location to an IP address, based on a database. `vmod_mmdb` came into play when `libgeoip`, the library that `vmod_geoip` depended on, was deprecated.

The previous database format had also been deprecated. `vmod_mmdb` supports the new `libmaxminddb` library, and the new database format that comes with it. As a bonus, this new geolocation module can retrieve a lot more information from an IP address than just the country information.

Here's a VCL example in which we retrieve both the country and city information from the client IP address:

```
vcl 4.1;

import mmdb;

sub vcl_init {
    new geodb = mmdb.init("/path/to/database");
}

sub vcl_recv {
    return(synth(200,
        "Country: " + geodb.lookup(client.ip, "country/names/en") +
" - " +
        "City: " + geodb.lookup(client.ip, "city/names/en")
    ));
}
```

MaxMind, the company that provides the library and the database, has a free offering. However, retrieving more detailed information will probably require a commercial license.

vmod_utils

As the name indicates, `vmod_utils` is a *utility module* that groups features that are too small to deserve their own *VMOD*.

Without any further ado, let's just dive into a couple of examples:

The following example uses `utils.newline()` to print a new line between `first line` and `second line`:

```
vcl 4.1;

import utils;

sub vcl_recv {
    return(synth(200, "first line" + utils.newline() + "second
line"));
}
```

The next example prints a timestamp in a specific date-time format. It is basically a wrapper around the `strftime` function in the *C language*:

```
vcl 4.1;

import utils;

sub vcl_recv {
    return(synth(200, utils.time_format("%A %B%e %Y")));
}
```

The output will be something like `Monday June 8 2020`.
Another useful example leverages `utils.waitinglist()` and `utils.backend_misses()` to ensure we only cache objects on the *second miss:*

```
vcl 4.1;

import utils;

sub vcl_backend_response {
    if (!utils.waitinglist() && utils.backend_misses() == 0) {
        set beresp.uncacheable = true;
        set beresp.ttl = 24h;
    }
}
```

When `utils.backend_misses()` is `0`, the object is not yet in *hit-for-miss* status. This allows us to make it uncacheable until the next time a miss happens, in which case `utils.backend_misses()` will be `1`.

But by adding the `!utils.waitinglist()` check, we make sure we only trigger a *hit-for-miss* when no other requests are on the waiting list. Otherwise these requests would not benefit from *request coalescing* and *request serialization* would occur.

Although `vmod_utils` has plenty more functions, here's just one more example. It's the `utils.dyn_probe()` function that creates a dynamic probe.

Dynamic probes are not predefined using a `probe {}` construct, but can be set *on-the-fly*. They are used to probe *dynamic backends*. Just like these dynamic probes, dynamic backends aren't predefined using a `backend {}` construct, but can be set *on-the-fly*.

```
vcl 4.1;

import utils;
import goto;

backend default none;

sub vcl_init {
    # set the probe URL to perform health checks
    new dyn_probe = utils.dyn_probe(url="/");
    # assign dynamic probe to dynamic backend definition
    new dyn_dir = goto.dns_director("example.com", probe=dyn_probe.
probe());
}

sub vcl_backend_fetch {
    set bereq.backend = dyn_dir.backend();
}
```

Explicitly return errors

With `return(error())`, you can explicitly return an error in both `vcl_backend_fetch` and `vcl_backend_response` subroutines. This immediately takes you into the `vcl_backend_error` state, where the error message is returned to the client.

Here's an example where the error is returned before a backend request is sent:

```
vcl 4.1;

sub vcl_backend_fetch {
    return(error(500,"Something is wrong here"));
}
```

Here's an example where the error is returned after a backend response was received. Even if the backend response turned out to be successful, we can still decide to return an error:

```
vcl 4.1;

sub vcl_backend_response {
    return(error(500,"Something is wrong here"));
}
```

JSON formatting support in varnishncsa

`varnishncsa` is a program that ships with Varnish and that returns *access logs* in an *NCSA format*. This is pretty much the standard format that most web servers use.

The new `-j` flag doesn't convert the output into *JSON*, as you might expect. It actually makes the output *JSON safe*.

Here's an example of `varnishncsa` output without any extra formatting:

```
$ varnishncsa
172.18.0.1 - - [08/Jun/2020:15:47:46 +0000] "GET http://localhost/
HTTP/1.1" 200 0 "-" "curl"
```

When we actually create a custom output format using `-F` that looks like *JSON*, the `-j` option will make sure the output is *JSON safe*. Here's an example:

```
$ varnishncsa -j -F '{ "received_at": "%t", "response_bytes": %b,
"request_bytes": %I, "time_taken": %D, "first_line": "%r", "status":
%s }'
{ "received_at": "[08/Jun/2020:16:03:33 +0000]", "response_bytes":
5490, "request_bytes": 472, "time_taken": 155, "first_line": "GET
http://localhost/ HTTP/1.1", "status": 200 }
```

2.3.3 vmod_str

`vmod_str` is a string *VMOD* that contains a collection of *helper functions*. I'll highlight a couple of these helper functions in a single *VCL* example:

```
vcl 4.1;

import str;

sub vcl_recv {
    set req.http.x-str = "vmod_str functions example";

    if (str.len(req.http.x-str) <= 0) {
        return(synth(500,"String is empty"));
    }

    if (str.contains(req.http.x-str," ")) {
        set req.http.x-output = str.split(req.http.x-str,1," ");
    } else {
        set req.http.x-output = req.http.x-str;
    }

    set req.http.x-output = str.reverse(req.http.x-output);

    return(synth(200,req.http.x-output));
}
```

Here's a quick rundown of what this *VCL* file does:

- The input string we are inspecting and modifying contains `vmod_str functions example`
- If this input string is empty, or the header is not set, returns an error
- If the input string contains spaces, splits the string on spaces, and keeps only the first word
- Otherwise just uses the input string as-is
- Reverses the characters and returns the string

In our case, the output will be `rts_domv`, which is the reverse string of `vmod_str`.

vmod_mse

`vmod_mse` gives users control over how *MSE* behaves on a *per request basis* in *VCL*. Although *MSE* works fine without this *VMOD*, it does offer fine-grained control to users that require it.

This *VMOD* has two functions:

- `mse.set_weighting()` : set the algorithm that is used for filling stores
- `mse.set_stores()` : selects one or more stores that match a given tag

Set weighting algorithm

By setting the weighting algorithm, *MSE* will switch from basic *round robin* store selection, to *weighted round robin*. The example below uses the *store size* as the weight:

```
vcl 4.1;

import mse;

sub vcl_backend_response {
    mse.set_weighting(size);
}
```

This means that *MSE* will store more objects in the stores that have more space, rather than in the smaller stores. The effect is that different-sized stores will become full, and least recently used nuking will start, roughly at the same time.

By setting `mse.set_weighting(available);`, *MSE* will give store more objects in stores that have most space available.

When setting `mse.set_weighting(smooth);`, *MSE* will combine *store size* and *available store space* to come up with a weight.

Select stores by tag

In order to select stores by tag, the stores will need to be tagged in the *MSE configuration file*. Here's such a file:

```
env: {
    id = "myenv";
    memcache_size = "100GB";

    books = ( {
        id = "book1";
        directory = "/var/lib/mse/book1";
        database_size = "1G";

        stores = ( {
            id = "store1";
            filename = "/var/lib/mse/stores/disk1/store1.dat";
            size = "1T";
            tags = "sata";
        }, {
            id = "store2";
            filename = "/var/lib/mse/stores/disk2/store2.dat";
            size = "1T";
```

```
                tags = "ssd";
            } );
        });
        default_stores = "none";
};
```

This configuration has two stores, each with their own tag. In this case, the tags represent the type of disk they're hosted on: *store 1* has a slower SATA disk and is tagged as `sata`. *store 2* has a faster SSD disk and is tagged as `ssd`.

If no tag is explicitly set, no store will be selected, and the objects will be stored in memory only. This behavior is the result of `default_stores = "none";`.

Once the tags have been assigned to their corresponding store, we can start setting tags for specific content, as illustrated in the *VCL* example below:

```
vcl 4.1;

import mse;
import std;

sub vcl_backend_response {
    if (beresp.ttl < 120s) {
        mse.set_stores("none");
    } else {
        if (beresp.http.Transfer-Encoding ~ "chunked" ||
        std.integer(beresp.http.Content-Length,0) > std.bytes("1M"))
{
            mse.set_stores("sata");
        } else {
            mse.set_stores("ssd");
        }
    }
}
```

This *VCL* file will keep short-lived content with a *TTL* of less than *120 seconds* in memory only.

When an object has a *TTL* that is greater than two minutes, it will be persisted to disk. Based on the content size, the *VCL* file will either set the `sata` tag, or the `ssd` tag.

In this case, Varnish will store objects with a content size bigger than *1 MB* on *MSE stores* that are tagged with the `sata` tag. The same applies when there's no `Content-Length` header and *chunked transfer encoding* is used.

All other content will be stored on *MSE stores* that are tagged with the `ssd` tag.

Last byte timeout

When configuring timeouts for backend interaction, the *first byte timeout* is a very common parameter to tune: it represents the amount of time Varnish is willing to wait before the backend sends the first byte.

The *time to first byte* indicates how fast, or how slow, a backend application is able to respond.

With the introduction of the *last byte timeout*, Varnish can now decide how long it is willing to wait for the last byte to be received. When this timeout occurs, it usually means there is network latency. But when *chunked transfer encoding* is used, it can mean that it takes too long for the last chunk to be sent and received.

You can either define this using a backend definition in your *VCL* file:

```
backend default {
    .host = "localhost";
    .port = "8080";
    .last_byte_timeout = 90s;
}
```

Or you can explicitly set this value in your `vcl_backend_fetch` logic:

```
vcl 4.1;

sub vcl_backend_fetch {
    set bereq.last_byte_timeout = 90s;
}
```

The code above overrides any timeout set in the backend definition. It also overrides the default timeout, which is 0, meaning it will wait forever. The default timeout can be changed with the `last_byte_timeout` runtime parameter. Like all runtime parameters, the change can be made persistent by adding it to the command line in your service file:

```
ExecStart=/usr/sbin/varnishd ... -p last_byte_timeout=90
```

If-Range support

Range requests are a common HTTP feature, which Varnish supports as well.

By issuing a `Range` header upon request, a client can request a selected range of bytes to be returned, instead of the full body. The response will not have an `HTTP 200 OK` status, but an `HTTP 206 Partial Content` status code.

Varnish Enterprise 6 now also supports the `If-Range` request header in conditional fetches. This means when a range request is satisfiable, and the `If-Range` request header contains a value that matches either the `Etag` response header, or the `Last-Modified` response header, the range will be returned with an `HTTP 206 Partial Content` status code.

If the `If-Range` header doesn't match any of these response headers, the full response body is returned with an `HTTP 200 OK` status code.

Built-in TLS support

Up until *Varnish Enterprise 6.0.6r2*, no version of Varnish has ever had native support for *TLS*: neither *Varnish Enterprise*, nor *Varnish Cache*. The replacement for built-in TLS has been to use a TLS terminator like Hitch, which communicates connection meta information to Varnish through the *PROXY protocol*.

Varnish Enterprise now has *native TLS support*, which largely eliminates the need for a TLS proxy. TLS configuration is done in a separate file, as illustrated below:

```
frontend = {
    host = "*"
    port = "443"
}

pem-file = "/etc/varnish/certs/example.com"
```

That configuration file is then linked to the `varnishd` process by using the `-A` command line parameter. Here's an example:

```
ExecStart=/usr/sbin/varnishd ... -A /etc/varnish/tls.cfg
```

The TLS configuration syntax is the same that *Hitch* uses. *Hitch* is a dedicated *TLS proxy*, developed by *Varnish Software*. The configuration syntax is the same, which allows for a seamless transition from *Hitch* to native TLS.

Memory governor

The *memory governor* is a new feature of the *Massive Storage Engine*.

Correctly sizing the memory of your Varnish server can be tricky at times, especially at high scale. Besides the size of the payload in your cache, other support data structures and temporary *workspaces* for handling requests use a varying amount of memory. In addition to this, *transient storage* for *short-lived* objects is also allocated separately.

Even if only 80% of the server's memory is allocated for *cache payload data*, the server can still run out of memory when a large number of clients is connected, or when the individual objects are small.

The *memory governor* alleviates this issue by automatically adjusting the size of the cache based on the size of other allocations needed to run Varnish comfortably. The *memory governor* allows *MSE* to limit the memory of the `varnishd` process, instead of just the size of the payload data.

The *memory governor* is enabled by setting *MSE's* `memcache_size` property to *auto*:

```
env: {
    id = "myenv";
    memcache_size = "auto";
};
```

The `memory_target` runtime parameter limits the total size of the `varnishd` process. Its default value is *80%*. You can set the value to percentages, or absolute memory sizes.

vmod_jwt

`vmod_jwt` is a module for creating, manipulating, and verifying *JSON Web Tokens* and *JSON Web Signatures*.

JWT is a standard that APIs use to create and verify access tokens: the API creates the token, and the client sends it back to the API on every subsequent request to identify itself.

The *JWT* contains a set of public claims that the API can use. In order to guarantee the integrity of a token, the *JWT* also contains an *HMAC signature* that the API verifies.

`vmod_jwt` can verify incoming *JSON Web Tokens*, as illustrated in the example below:

```
vcl 4.1;

import jwt;

sub vcl_init {
    new jwt_reader = jwt.reader();
}

sub vcl_recv {
    if (!jwt_reader.parse(regsub(req.http.Authorization,"^Bearer
",""))) {
        return (synth(401, "Invalid JWT Token"));
    }

    if (!jwt_reader.set_key("secret")) {
        return (synth(401, "Invalid JWT Token"));
    }

    if (!jwt_reader.verify("HS256")) {
        return (synth(401, "Invalid JWT Token"));
    }
}
```

This example will look for an `Authorization` request header that matches *Bearer*, and then performs the following tasks:

- The token is extracted and parsed
- The secret key is set
- The token is verified using a *SHA256 HMAC signature*

`vmod_jwt` is also able to issue *JWTs*. In the next example we're issuing a token with the following properties:

- A *SHA256 HMAC signature* with `secret` as the secret key
- The subject is `1234567890`
- The issuer is `John Doe`
- The token is stored in the `token` response header

And here's the code to issue the *JWT* using `vmod_jwt`:

```
vcl 4.1;

import jwt;

sub vcl_init {
    new jwt_writer = jwt.writer();
}

sub vcl_backend_response {
    jwt_writer.set_alg("HS256");
    jwt_writer.set_sub("1234567890");
    jwt_writer.set_iss("John Doe");
    set beresp.http.token = jwt_writer.generate("secret");
}
```

vmod_stale

The new VMOD `stale` can be used to cleanly implement *stale if error* behavior in Varnish. Varnish's built-in *grace mode* covers the more well-known *stale while revalidate* caching behavior, but does not work well to implement a *stale if error*.

Stale while revalidate will keep stale content around while Varnish asynchronously gets the most recent version of the object from the cache. This can result in outdated content being served briefly.

This *VMOD* allows you to have a *stale if error* without *stale while revalidate*. This means: never serve outdated content, except when the backend is down.

The example below will revive a stale object when the backend returns a status code greater than or equal to 500:

```
vcl 4.1;

import stale;

sub stale_if_error {
    if (beresp.status >= 500 && stale.exists()) {
        # Tune this value to match your traffic and caching patterns
        stale.revive(20m, 1h);
        stale.deliver();
        return (abandon);
    }
}

sub vcl_backend_response {
    set beresp.keep = 1d;
    call stale_if_error;
}

sub vcl_backend_error {
    call stale_if_error;
}
```

This `stale.revive()` function will set a new *TTL* and a new *grace value*, as long as the total remaining lifetime of the object *(TTL + grace value + keep value)* is not exceeded. If any of these values exceed the total lifetime, the maximum remaining lifetime is used for either the *TTL* or the *grace value*, and the rest flows over into the *keep value*.

More information about the object lifetime, *time to live*, *grace*, and *keep* can be found in *chapter 3*.

vmod_sqlite3

SQLite is a library that implements a serverless, self-contained relational database system. As the name indicates, it's very lightweight, and relies on a single database file.

`vmod_sqlite` uses this library to offer *SQLite* support in Varnish. The following example uses `vmod_sqlite` to display the user's name based on an `ID` *cookie*:

```
vcl 4.1;

import sqlite3;
import cookieplus;

backend default none;

sub vcl_init {
    sqlite3.open("sqlite.db", "|;");
    sqlite3.exec({"CREATE TABLE `users` (`name` VARCHAR(100));"});
    sqlite3.exec("INSERT INTO `users` VALUES (‘John’), (‘Marc’),
(‘Charles’),(‘Mary’);");
}

sub vcl_fini {
    sqlite3.close();
}

sub vcl_recv {
    unset req.http.userid;
    unset req.http.username;
    cookieplus.keep("id");
    cookieplus.write();
    if (cookieplus.get("id") ~ "^[0-9]+$") {
        set req.http.userid = cookieplus.get("id");
        set req.http.username = sqlite3.exec("SELECT `name` FROM `us-
ers` WHERE rowid=" + sqlite3.escape(req.http.userid));
    }
    if (!req.http.username || req.http.username == "") {
        set req.http.username = "guest";
    }
    return(synth(200,"Welcome " + req.http.username));
}
```

As you can see, the `vcl_init` subroutine will initialize the *SQLite* database that is stored in the `sqlite.db` file. Various SQL statements are executed to create the `users` table, and to populate it.

When receiving requests, the value of the `ID` cookie is used as the *row identifier* of the record we're trying to fetch.

If there's a match, the username is returned, if not, *guest* is returned.

Take for example the following *HTTP* request:

```
GET / HTTP/1.1
Cookie:id=2
```

The output in this case will be `Welcome Marc`, because *row id 2* in the `users` table corresponds to `Marc`.

2.3.4 vmod_tls

Varnish Enterprise 6.0.6r2 features support for *native TLS*, meaning that TLS connections can be handled by Varnish, instead of depending on external *TLS termination*.

Up until *6.0.r2*, we had to use `vmod_proxy` to get TLS information for a TLS connection that was terminated elsewhere. Now, we can now use `vmod_tls` to get information about *native TLS connections*.

All sorts of TLS information is available:

- The TLS version
- The TLS cipher
- *Server Name Indication (SNI)* information

- *Application Layer Protocol Negotiation (ALPN)* information
- The certificate signature algorithm
- The certificate generation algorithm

But most importantly, there's a. `is_tls()` function that will return a boolean value to indicate whether or not the connection was made over TLS. Here's some example code:

```
vcl 4.1;

import tls;

sub vcl_recv {
    if (!tls.is_tls()) {
        set req.http.location = "https://" + req.http.host + req.url;
        return(synth(750,"Moved Permanently"));
    }
}

sub vcl_synth {
    if (resp.status == 750) {
        set resp.status = 301;
        set resp.http.location = req.http.location;
        return(deliver);
    }
}
```

This example is a typical use case where connections over *plain HTTP* automatically get redirected to the *HTTPS equivalent*.

vmod_headerplus

`vmod_headerplus` is an advanced header creation, manipulation, and introspection module. It facilitates tasks that otherwise would be done with more complicated regular expressions.

This *VMOD* allows easy access to complete header values, and specific attributes of a header value.

The first example shows how multiple headers that match a given pattern can be deleted using `headerplus.delete_regex()`:

```
vcl 4.1;

import headerplus;

sub vcl_recv {
    headerplus.init(req);
    headerplus.delete_regex("^X-");
    headerplus.write();
}
```

The example will delete all headers that start with `X-`.

The next example will look at the `Cache-Control` response header, and will set the `max-age` attribute to *20 seconds* if there's an `s-maxage` attribute:

```
vcl 4.1;

import headerplus;

sub vcl_backend_response {
    headerplus.init(beresp);
    if (headerplus.attr_get("Cache-Control", "s-maxage") {
        headerplus.attr_set("Cache-Control", "max-age","20");
    }
    headerplus.write();
}
```

vmod_resolver

In *Varnish Enterprise 6.0.6r8* `vmod_resolver` was added. It's a module that performs *Forward Confirmed reverse DNS (FCrDNS)*. This means that a *reverse DNS* resolution is done on the *client IP address*.

The resulting hostname is then resolved with a *forward DNS* resolution, and if any of the resulting IP addresses match the original *client IP address*, the check succeeds.

Here's some *VCL* to illustrate the feature:

```
vcl 4.1;

import std;
import resolver;

sub vcl_recv {
    if (resolver.resolve()) {
        std.log("Resolver domain: " + resolver.domain());
    } else {
        std.log("Resolver error: " + resolver.error());
    }
}
```

Veribot

Veribot is a *Varnish Enterprise 6.0.6r8* feature that leverages `vmod_resolver`. The goal of the *Veribot* feature is to check whether or not a *client* is a bot.

It does a first pass with a *User-Agent filter*, then a second pass using *FCrDNS*, which is performed by `vmod_resolver`.

Here's a *VCL example* of *Veribot*:

```
vcl 4.1;

include "veribot.vcl";

sub vcl_init {
    vb_ua_filter.add_rules(string = {"
        "(?i)(google|bing)bot"
        "(?i)slurp"
    "});

    vb_domain_rules.add_rules(string = {"
        ".googlebot.com" "allow"
        ".google.com" "allow"
        ".bingbot.com" "allow"
        ".slurp.yahoo.com" "allow"
        ".fakebot.com" "deny"
    "});
}

sub vcl_recv {
    call vb_check_client;
    if (req.http.vb-access != "allow") {
        return(synth(403,"Forbidden"));
    }
}
```

If the `User-Agent` header of a client passes the filter, the *FCrDNS* check succeeds, and the resulting domain name is allowed access by the domain rules, the client will be allowed access to the content.

This example assumes that the content would otherwise be guarded from regular users with a paywall. Verified bots, on the other hand, do get access to the content, and will index it for *SEO* purposes.

2.3.5 vmod_brotli

`vmod_brotli` was released in *Varnish Enterprise 6.0.6r10*, and offers *Brotli compression*.

Not only can `vmod_brotli` compress an object in cache using its optimized compression algorithm, it can also send *Brotli compressed responses* to browsers that support it.

When *Brotli compressed responses* are sent, the `Content-Encoding: br` header is also added.

Content from backends that send *native Brotli* to *Varnish*, will also be processed by *Varnish*, and stored in cache in that format.

Here's some *VCL code* to show you how to enable *Brotli* support:

```
vcl 4.1;

import brotli;

sub vcl_init {
    brotli.init(BOTH, transcode = true);
}

sub vcl_backend_response {
    if (beresp.http.content-encoding ~ "gzip" ||
        beresp.http.content-type ~ "text") {
        brotli.compress();
    }
}
```

2.3.6 vmod_format

String interpolation is possible in *Varnish* by closing the string, using the plus sign, and re-opening the string. When many values need to be included, this can become tedious.

`vmod_format` leverages the *ANSI C* `printf()` capabilities to easily perform string interpolation based on ordered arguments.

Imagine the following *VCL snippet*:

```
vcl 4.1;

sub vcl_synth {
    set resp.body = "ERROR: " + resp.status +"\nREASON: " + resp.rea-
son + "\n";
    return (deliver);
}
```

It's not that complicated, but we still have to open and close the string to insert the values into the string. As the size of the string grows, and the number of interpolated variables increases, things get more complicated.

The following *VCL example* uses `vmod_format` to make this task a lot easier:

```
vcl 4.1;

import format;

sub vcl_synth {
    set resp.body = format.quick("ERROR: %s\nREASON: %s\n",
        resp.status, resp.reason);
    return (deliver);
}
```

2.3.7 scoreboard

The `varnishscoreboard` program, which was introduced in *Varnish Enterprise 6.0.4r3*, and redesigned for *Varnish Enterprise 6.0.7r1*, displays the current state of the various active threads.

Here's some example output from this tool:

```
$ varnishscoreboard
Age       Type      State         Transaction Parent      Address
Description
   1.64s probe    waiting                 0           0 -
boot.default
   2.11m acceptor accept                  0           0 :6443
a1
   2.11m acceptor accept                  0           0 :6443
a1
   2.11m acceptor accept                  0           0 :80
```

```
a0
   0.03s acceptor accept                  0            0 :80
a0
   0.01s backend  startfetch         360910       360909 -
POST example.com /login
   0.01s client   fetch              360909       360908
172.19.0.1:63610    POST example.com /login
   2.11m acceptor accept                  0            0 :6443
a1
   2.11m acceptor accept                  0            0 :6443
a1
   2.11m acceptor accept                  0            0 :80
a0
   0.01s acceptor accept                  0            0 :80
a0
Threads running/in pool: 10/90
```

We'll cover `varnishscoreboard` in more detail in *chapter 7*.

2.3.8 Features ported from Varnish Cache Plus 4.1

Since the release of *Varnish Enterprise 6* in 2018, a lot of new features have been added. But the very first features that were added were actually ported from *Varnish Cache Plus 4.1*. Users were quite accustomed to a range of features, and didn't want to lose them.

The *Varnish Software* team ported the following enterprise features:

- Parallel ESI
- Edgestash
- Dynamic backends
- Backend TLS
- Key-value store
- Least connections director
- Real-time status module
- Varnish High Availability
- `vmod_aclplus`
- `vmod_cookieplus`

- `vmod_http`
- `vmod_rewrite`
- `vmod_session`

And the following open source *VMODs* were also packaged:

- `vmod_bodyaccess`
- `vmod_cookie`
- `vmod_header`
- `vmod_saintmode`
- `vmod_tcp`
- `vmod_var`
- `vmod_vsthrottle`
- `vmod_xkey`

Some of these open source *VMODs* have since been replaced with an enterprise equivalent, but are still available for backwards compatibility reasons.

2.3.9 What happens when a new Varnish Cache version is released?

As mentioned before, *Varnish Enterprise 6* is built on top of *Varnish Cache 6.0*, so it follows its releases. These releases are *patch releases*, so they usually relate to bugfixes or security updates.

When the open source community notices a bug, and it gets fixed in *Varnish Cache*, the fix will be ported to *Varnish Enterprise* as well, and the *patch version number* increases.

On February 4th 2020 for example, a security vulnerability was fixed in *Varnish Cache 6*. This resulted in the immediate release of version *6.2.3*, *6.3.2*, and *6.0.6 (LTS)*. Naturally, a new version of *Varnish Enterprise* was released and became version *6.0.6r1*.

It also works the other way around: sometimes an important fix happens in *Varnish Enterprise* first, which results in a new release. Then the fix is ported to *Varnish Cache*, which results in a release as well. Because this new *Varnish Cache* release doesn't add anything new at that point, the subsequent *patch release* of *Varnish Enterprise* is postponed until there's enough new additions and fixes to warrant another release.

Because *Varnish Enterprise 6* follows the *Varnish Cache 6.0* release schedule, new features in other *Varnish Cache* minor versions are not automatically added to *Varnish Enterprise*. Feature porting from *6.1*, *6.2*, *6.3*, *6.4*, *6.5* and *6.6* happens on a case-by-case basis. Compatibility breaking changes, however subtle they may be, are not ported.

2.4 Where to get it

Installing some version of *Varnish* is quite easy: your Linux distribution's package managers probably have a *Varnish Cache* package available. You can even compile Varnish from source, and follow the development of the project, if you wish.

There are a lot more ways to get a hold of *Varnish*, both *Varnish Cache* and *Varnish Enterprise*. The official packages by *Varnish Software* will give you the recommended version of *Varnish*.

2.4.1 The official package repositories

The recommended official package repositories are hosted on Packagecloud, and they're available for Debian-based systems *(Debian and Ubuntu)*, and for RPM-based systems *(Red Hat, CentOS, Fedora)*, both for *Varnish Cache*, and for *Varnish Enterprise*.

For *Varnish Cache*, you can find these packages on https://packagecloud.io/varnish-cache. You can either pick individual releases, or one of the weekly builds. Since these packages are supported for a limited time, my advice is to just install Varnish 6.0 LTS.

> There is an equivalent for *Varnish Enterprise*, but it requires an *access token*, which comes with the *Varnish Enterprise* license.

Using *Packagecloud* repositories is quite easy. For *Debian-based* systems, you can run the following command to set up the repository on your machine:

```
curl -s https://packagecloud.io/install/repositories/varnishcache/
varnish60lts/script.deb.sh | sudo bash
```

This snippet will download and execute a Bash script, that detects which Linux distribution you have. It provisions the right channels, and you can run `apt-get install varnish` to install Varnish using our official packages.

There is also a manual way of configuring the repositories, if you are uncomfortable running a Bash script you have downloaded from the internet as the root user.

For *RPM-based* systems, you'll have to run the following equivalent:

```
curl -s https://packagecloud.io/install/repositories/varnishcache/
varnish60lts/script.rpm.sh | sudo bash
```

And you install Varnish using our official packages by running `yum install varnish`

2.4.2 Installing from source

Varnish Cache is open source, so yes, you can access and download the source.

The Releases & Downloads page on the Varnish Cache website provides every release, from *Varnish Cache 0.9* all the way up to *Varnish Cache 6.6*.

The development of *Varnish Cache* happens on GitHub. You can look at individual commits, issues, pull requests, and various branches. The releases page also offers all released versions of *Varnish Cache*, as well as beta versions, and release candidates. If you do not want to install from source, the weekly packages provide a convenient way to check out the latest developments of *Varnish Cache*.

2.4.3 Official Docker image

There is an official Docker image for *Varnish Cache* on the Docker Hub.

By running the following command, you spin up a Docker container using the latest version of *Varnish Cache*, which currently is *version 6.6*:

```
docker run --name my-running-varnish -v /path/to/default.vcl:/etc/
varnish/default.vcl:ro --tmpfs /var/lib/varnish:exec -d -p 8080:80
varnish:latest
```

This command will:

- Run the `latest` tag of the `varnish` Docker image
- Name the container `my-running-varnish`
- Mount the hypothetical VCL file `/path/to/default.vcl` onto `/etc/varnish/default.vcl` in the container
- Mount the `/var/lib/varnish` folder as a temporary file system in memory with `exec` privileges
- Forward the container's *port 80* to the host's *port 8080*

At this point the `latest` tag on the `varnish` Docker image refers to version `6.6`, which can also be used as a tag. Another tag that refers to the same version is `fresh`. That's what we call the latest *Varnish Cache* release.

However, you can also run a *Varnish 6.0 LTS* Docker container by using either the `6.0` tag, or the `stable` tag.

In summary, these are the tags of the official *Varnish Cache* Docker image that are currently available:

- `varnish:latest` *(version 6.6.0)*
- `varnish:fresh` *(version 6.6.0)*
- `varnish:6` *(version 6.6.0)*
- `varnish:6.6` *(version 6.6.0)*
- `varnish:6.6.0` *(version 6.6.0)*
- `varnish:stable` *(version 6.0.8)*
- `varnish:6.0` *(version 6.0.8)*
- `varnish:6.0.8` *(version 6.0.8)*

2.4.4 Official cloud images

A collection of official Varnish images is available in the respective marketplace of cloud providers like:

- Amazon Web Services *(AWS)*
- Microsoft Azure
- Google Cloud Platform *(GCP)*
- Oracle Cloud Infrastructure *(OCI)*
- DigitalOcean

These images contain pre-installed versions of Varnish with the required configuration to get started immediately.

AWS, *Azure*, *GCP*, and *OCI* offer *Varnish Enterprise* images, whereas DigitalOcean currently only offers *Varnish Cache* images.

Varnish Enterprise features in the cloud

The *Varnish Enterprise* image also comes with pre-installed and pre-configured tools like:

- *Hitch* to terminate TLS connections
- *Varnish Broadcaster* to broadcast individual requests to a group of Varnish servers
- *Varnish Discovery*, which automatically keeps the inventory of your Varnish cluster up-to-date based on autoscaling groups
- *Varnish Agent*, which allows remote management of your server
- *VCS Agent* to send custom statistics from individual Varnish servers to a *Varnish Custom Statistics* server that centralizes stats

Licensing and billing

What makes this cloud offering so interesting is that users can try *Varnish Enterprise* without having to buy a license upfront: there is an hourly license cost associated with running these images, which the cloud provider bills on top of their own service charges.

You're not interacting with *Varnish Software* directly, because your cloud platform's marketplace will act as a broker.

The licensing model for *AWS*, *Azure*, and *GCP* is flexible in such a way that you can spin up a *Varnish Enterprise* virtual machine using our official images for a couple of hours, and end up spending only $1 on licensing.

Oracle's *OCI* platform applies a *bring your own license* model, which works well for enterprise users.

However, if you don't buy a license with *Varnish Software*, you're not entitled to the same level of service and support that *Varnish Software* clients enjoy.

On *DigitalOcean* there is no license fee because *Varnish Cache* is an open source project. The only bill you will get there is your infrastructure bill.

2.5 Chapter summary

Go Varnish 6 or go home!: that's pretty much the message of this chapter.

Varnish has a big responsibility in terms of acceleration, stability, and content delivery. We cannot emphasize enough that being on a supported version is crucial.

Varnish Cache 6.0 LTS or *Varnish Enterprise 6* are the versions you should use, and at this point, the reason why should be crystal clear.

We're already two chapters into this book. We're past the introduction phase, and all the disclaimers have been presented. You know what Varnish is; you know what it does. Now it's time to start using it.

In *chapter 3*, we'll be looking at how you can control Varnish's behavior using conventional *HTTP caching mechanisms*.

Chapter 3: It's all about HTTP

HTTP is not just one of the things we do, it's the *only* thing we do.

In this chapter we will talk about *HTTP's built-in caching semantics*, and how they allow you to control the behavior of Varnish.

When it comes to configuring Varnish, and customizing the behavior of the cache, the *Varnish Configuration Language* seems like the best choice. The level of flexibility you get seems unparalleled.

But here's a statement that might challenge that idea:

Write as little VCL as possible; let HTTP do the heavy lifting.

It looks like we're undercutting the number one feature of Varnish, but there's a bit more nuance to the statement:

- The HTTP protocol has *built-in caching semantics*.
- These caching mechanisms are both officially specified and widely adopted.
- Using the right *HTTP response* headers, backends can express whether a page can be cached, how, and for how long.
- Varnish can automatically parse and act upon those headers, giving more control to the backends.

By leveraging *HTTP headers*, application developers have a conventional and portable way to control the behavior of the cache. Even if Varnish is swapped out for another caching proxy, the behavior should be the same.

Although you can achieve a lot more by writing VCL, using HTTP can reduce the line count of your VCL file, making it easier to maintain. It also make sense from an architectural point of view to let the backend, which is the content producer, decide how long said content can be cached.

So without further ado, let's talk about leveraging HTTP.

3.1 HTTP as the go-to protocol

There is a difference between *the internet* and the *World Wide Web*: the internet offers us a multitude of protocols to interact with computers over a global network.

The *World Wide Web* is a specific application of the internet that depends on the HTTP protocol and its siblings. This protocol has always been the engine behind web pages and behind hypermedia, but HTTP has grown and can do so much more now.

Although traditional *client-server interactions* over HTTP using a web browser are still very common, it's the fact that machines can communicate with each other over HTTP that took the protocol to the next level.

APIs, *service-oriented architectures*, *remote procedure calls*, *SOAP*, *REST*, microservices. Over the years, we've seen many buzzwords that describe this kind of *machine-to-machine communication*.

Why develop a custom protocol? Why re-invent the wheel, when HTTP is so accessible?

3.1.1 The strengths of HTTP

HTTP is actually an implementation of the Representational State Transfer (REST) architectural style for distributed hypermedia systems.

We know, it's quite the mouthful. The design of *REST* came out of Roy Thomas Fielding's Ph.D. dissertation, and many of the strengths of HTTP are described in that chapter of the dissertation.

HTTP is a pretty simple stateless protocol that is *request-response* based. There is a notion of *resources* that can reflect entities of the application's business logic. Resources can be identified through *URLs*, and can be represented in various document and file formats.

The type of action that is performed through an HTTP request is very explicit: because of *request methods*, the intent is always clear. A `GET` request is used for *data retrieval*, a `POST` request is all about *data insertion*. And there are many more valid *HTTP request methods*, each with their own purpose.

The real power of HTTP lies in its *metadata*, which is exposed through *request and response headers*. This metadata can be processed by clients, servers, or proxies, and improve the overall experience.

Some of the metadata is considered *hypermedia*, helping users navigate through resources, presenting resources in the desired format. In essence, it is what makes the *World Wide Web* so interactive and so impactful.

What makes Roy Thomas Fielding's dissertation so relevant to our use case, which is web acceleration and content delivery, can be summarized in the following quote:

> REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

The entire purpose is to make applications scale and to reduce latency. The fact that caching is explicitly mentioned as one of the features of REST makes it a *first-class citizen*. It is also by design that so-called *intermediary components* exist.

Varnish is such an intermediary component, and Varnish reduces interaction latency through caching. A tool like Varnish, among others, facilitates the scalability of HTTP, and as a consequence, the scalability of the web.

3.1.2 The limitations of HTTP

HTTP is not a perfect protocol. Although it probably tackled most of the issues it was designed for, times change, and use cases evolve.

> In the nineties, I knew that HTTP was what powered the interaction between my Netscape browser and some web server. Today, in 2021, I know that HTTP is used by my cell phone when it connects to my car to see if the windows are closed. Now that's an example of evolving use cases.

HTTP is now used in situations it wasn't designed for. And although it's doing an okay job, there are some limitations:

- HTTP is stateless; we have to use cookies to keep track of state.
- There are mechanisms in place to define cache expiration *(through the `Cache-Control` header)*, but there is no conventional mechanism for explicit cache purging.
- The `Connection` header is used by clients to decide whether or not the connection should be closed after processing an HTTP request. Although it's a popular mechanism, proper use is at the discretion of the client and server.

- Long polling, and *Server-Sent Events* are used for event-based communication, whereas HTTP itself is not really suited for those situations.
- Although connections can be reused, concurrency requires opening up multiple connections to the server, but clients must be careful not to open too many to avoid overloading servers.
- Even though the payload of an HTTP response can be compressed with *gzip*, the headers remain uncompressed.

And there are many more limitations. Despite these limitations, and the fact that HTTP has outgrown its original purpose, we still trust HTTP as a *low entry barrier protocol*.

3.1.3 Newer versions of the HTTP protocol

Although the previous section portrayed the limitations of HTTP, we should emphasize that it has evolved over the years, and continues to do so.

In 1991, *Tim Berners-Lee* released HTTP as a single-line protocol, which bootstrapped the *World Wide Web*. Back then, HTTP didn't have a version number.

It wasn't until `HTTP/1.0` was released back in 1996, that its predecessor received the `HTTP/0.9` version number. `HTTP/1.0` was already a multi-line protocol and feature headers. In fact, it already looked a lot like the HTTP we know today.

HTTP/1.1

In 1997 `HTTP/1.1` was released. As the version number indicates, it added new features, but it wasn't a complete overhaul. Here are a couple of notable feature additions that were part of the `HTTP/1.1` release:

- The `Host` header is a required request header
- Support for persistent connections through `Connection: keep-alive`
- Support for the `OPTIONS` method
- Better support for *conditional requests* through the `Etag` and `If-None-Match` headers
- Advanced caching support through the `Cache-Control` header
- Cache variations support through the `Vary` header
- Streaming support using `Transfer-Encoding: chunked`

- Support for compressed responses through the `Content-Encoding` header
- Support for range requests

There are of course many more features in that release, but this overview shows that significant improvements were made. Even in the mid-nineties, when the web was still in its infancy, people had the impression that HTTP was used beyond its scope. HTTP had to improve, which it did, and still does to this day.

HTTP/2

In 2015, `HTTP/2` was officially released as the new major version of the protocol. `HTTP/2` was inspired by *SPDY*, a protocol invented by Google to improve transport performance and reduce latency.

In `HTTP/1.1`, only one request at a time could be sent over a single connection, and you needed for this request to be done to tackle the next one. This is referred to as *head-of-line blocking*. In order to benefit from concurrency, multiple TCP connections should be opened. This obviously has a major impact on performance and latency, and is further amplified by the fact that modern websites require an increasing amount of resources: JavaScript files, CSS files, web fonts, AJAX calls, and much more.

`HTTP/2` tackles these inefficiencies by enabling full request and response multiplexing. This means that one TCP connection can exchange and process multiple HTTP requests and responses at the same time.

The protocol became mostly binary. Messages, both requests and responses, were fragmented into frames. Headers and payload were stored in different frames, and correlated frames were considered a message. These frames and messages were sent over the wire on one or multiple streams, but all in the same connection.

This shift in the way message transport was approached resulted in fewer TCP connections per transaction, less *head-of-line blocking*, and lower latency. And fewer connections means fewer *TLS handshakes*, reducing overhead even further.

Another benefit of `HTTP/2` is the fact that headers can also be compressed. This feature was long overdue because payload compression was already quite common.

HTTP/3.0

With `HTTP/2` we significantly reduced latency by multiplexing requests and responses over a single TCP connection. This solves *head-of-line blocking* from an HTTP point of view, but not necessarily from a TCP point of view.

When packet loss occurs, there is *head-of-line blocking*, but it's at the TCP level. Even if the packet loss only occurs on a single request or response, all the other messages are blocked. TCP has no notion of what is going on in higher-level protocols, such as HTTP.

`HTTP/3.0` aims to solve this issue by no longer relying on TCP, but using a different transfer protocol: *QUIC*.

QUIC looks a lot like TCP, but is built on top of UDP. UDP has no *loss recovery* mechanisms in place and is a so-called *fire and forget* protocol. The fact that UDP has no handshaking allows for *QUIC* to multiplex without the risk of *head-of-line blocking*. Potential packet loss will only happen on the affected transaction, and will not block other transactions.

QUIC does implement a low-overhead form of handshaking that doesn't rely on the underlying protocol. As a matter of fact, TLS negotiation is also done in *QUIC* during the handshaking. This heavily reduces extra roundtrips, compared to TLS on top of TCP.

This new *QUIC* protocol is a very good match for HTTP, and moves a lot of the transport logic from the transport layer into user space. This allows for HTTP to be a lot smarter when it comes to transport and message exchange, and makes the underlying transport protocol a lot more robust.

What initially was called *HTTP over QUIC*, officially became `HTTP/3.0` in 2018. The way that HTTP header compression was implemented in `HTTP/2` turned out to be incompatible with `QUIC`, which resulted in the need to bump the major version of HTTP to `HTTP/3.0`.

Most web browsers offer `HTTP/3.0` support, but on the web server front, it is still early days. *LiteSpeed* and *Caddy* are web servers that support it, but there is no support for it in *Apache*, and *Nginx* only has a tech preview of `HTTP/3.0` available.

3.1.4 What about Varnish?

Varnish supports `HTTP/1.1` and `HTTP/2`. Requests that are sent using `HTTP/0.9`, or `HTTP/1.0` will result in an `HTTP/1.1` response.

As you will see in the next sections, Varnish will leverage many HTTP features to decide whether or not a response will be stored in cache, for how long it will be stored in cache, how it is stored in cache, and how the content will be delivered to the client.

Here's a quick preview of Varnish *default* behavior with regard to HTTP:

- Varnish will inspect the *request method* of an HTTP request, and only cache `GET` or `HEAD` requests.
- Varnish will not serve responses from cache where the request contains *cookies* or *authorization* headers.
- Varnish can serve compressed data to the client using *Gzip compression*. If the client doesn't support it, Varnish will send the plain text version of the response instead.
- Varnish will respect the `Cache-Control` header and use its values to decide whether or not to cache and for how long.
- The `Expires` header is also supported and is processed when there's no `Cache-Control` header in the response.
- The *Vary* header is used to support cache variations.
- Varnish supports conditional requests, both for clients and backends.
- Varnish uses the values of the `Etag` and `Lost-Modified` response headers and compares them to `If-None-Match` and `If-Modified-Since` request headers for conditional requests.
- The special `stale-while-revalidate` attribute from the `Cache-Control` header is used by Varnish to determine how long stale content should be served, while Varnish is revalidating the content.
- Varnish can serve *range requests* and supports *conditional* range requests by comparing the value of an `If-Range` header to the values of either an `Etag` header, or a `Last-Modified` header.
- Varnish supports content streaming through *chunked transfer encoding*.

> This is just default behavior. Custom behavior can be defined in *VCL* and can be used to leverage other parts of HTTP that are not implemented by default.

HTTP/2 in Varnish

Getting back to `HTTP/2`: Varnish supports it, but you need to add the following feature flag to enable support for *H2*:

```
-p feature=+http2
```

Because the browser community enforced *HTTPS* for *H2*, you need to make sure your *TLS proxy* has *H2* as valid *ALPN protocol*.

If you use *Hitch* to terminate your TLS connection, you can add the following value to your Hitch configuration file:

```
alpn-protos = "h2, http/1.1"
```

If you use a recent version of *Varnish Enterprise*, you can enable *native TLS support*, which will handle the *ALPN* part for you.

HTTP/3 in Varnish

`HTTP/3` is on the roadmap for both *Varnish Cache* and *Varnish Enterprise*, but the implementation is only in the planning stage for now, with no estimated time of delivery as the protocol itself hasn't been finalized yet.

The changes needed to support `HTTP/3` are substantial, and such changes will always warrant an increase of the major version number.

Basically, it will take at least until *Varnish 7* for `HTTP/3` to be supported in Varnish.

3.2 HTTP caching

The entire concept of caching is not an afterthought in HTTP. As the protocol evolved, specific headers were introduced to support caching.

However, there's always a difference between the specification and the implementation of the protocol.

Originally, HTTP's caching headers were designed for browser caching. But experience has taught us, this is not a reliable solution:

- The cache is not shared, because every client caches the data independently, so the server still needs to serve each client.
- Browser caching can easily by disabled or bypassed.

Luckily, reverse caching proxy servers like Varnish can also interpret these caching headers. These are *shared caches*, and HTTP even has specific semantics to control these sorts of caches.

> The fact that these headers exist allows for caching policies to become portable: they are part of the code, they are part of a conventional specification, and they should be respected by any kind of caching device or software. This reduces vendor lock-in while allowing developers to better express their intentions.

Let's have a look at these headers, and see how they allow you to cache.

3.2.1 The Expires header

The `Expires` response header isn't really an exciting header. It's also quite limited in its usage. Here's an example:

```
Expires: Wed, 1 Sep 2021 07:28:00 GMT
```

The idea is that a response containing this header can be stored in cache until *September first 2021 at 07:28 (GMT time zone)*. Once that time is reached, the cached object is considered stale and should be revalidated.

There's not a lot of nuance to it:

- A response with a date in the future can be cached.
- A response with a date in the past cannot be cached.

Although Varnish supports this header, it's not that common. You're better off using the `Cache-Control` header instead.

Some servers will convey that a response isn't cacheable by setting an `Expires` at the beginning of the Unix Time:

```
Expires: Thursday, 1 January 1970 00:00:00 GMT
```

`Expires` has been deprecated since HTTP/1.1 and should be avoided. If both a `Cache-Control` header and an `Expires` header are present, `Expires` is ignored.

3.2.2 The Cache-Control header

The `Cache-Control` header is the main tool in your toolbox when it comes to controlling the cache. Compared to `Expires`, the semantics of `Cache-Control` are a lot broader, as it is actually a list of finer-grained properties.

The primary expectation of any caching header is to indicate how long a response should be cached. Implicitly this also allows you not to cache certain responses. `Cache-Control` also has the capabilities to express what should happen when an object has expired.

The `Cache-Control` header is both a request and a response header. We commonly use it as a response header to describe how caching the response should be approached. But a browser can also send a `Cache-Control: no-cache` to indicate that it doesn't want to receive a cached version of a response.

max-age vs s-maxage

The first example features both `max-age` and `s-maxage`:

```
Cache-Control: s-maxage=86400, max-age=3600
```

`max-age` is aimed at browsers. In this example, a browser can cache the response for an hour, which corresponds to 3600 seconds because of the `max-age=3600` keyword.

`s-maxage`, however, is aimed at shared caches like Varnish. In this example, Varnish is allowed to cache the response for a day because of the `s-maxage=86400` keyword.

If Varnish sees the `s-maxage` keyword, it will take that value as the *TTL*. If there's no `s-maxage`, Varnish will use the `max-age` value instead.

Public vs private

The `public` keyword indicates that both shared and private caches *(browsers)* are allowed to store this response in cache. Here's an example:

```
Cache-Control: public, max-age=3600
```

In this example, both the browser and Varnish are allowed to cache the response for an hour.

The `private` keyword, on the other hand, prohibits shared caches from storing the response in cache:

```
Cache-Control: private, max-age=3600
```

The example above only allows browsers to cache the response for an hour.

Deciding not to cache

The `Cache-Control` header offers many ways of indicating that a response should not be cached:

```
Cache-Control: s-maxage=0
```

This example uses a *zero TTL* to keep a response from being cached. It also works with max-age:

```
Cache-Control: max-age=0
```

If you just want to avoid that a shared cache stores the response, issuing `private` will do:

```
Cache-Control: private
```

And then there's the well-known `no-cache` and `no-store` keywords:

```
Cache-Control: no-cache, no-store
```

- `no-cache` means that the data in cache shouldn't be used without a systematic revalidation: the agent always needs to verify that the cached version is the current one.
- `no-store` means that the object shouldn't even be stored in cache, let alone be served from cache.

> The `no-cache` and `no-store` keywords each have their own nuance, but most of the time they have the same effect, depending on the implementation.

Revalidation

When a cached object expires, it's up to the cache to revalidate the content with the origin server.

In its simplest form, a request to an expired object will trigger a synchronous backend fetch and will update the object.

Some implementations, including Varnish, support asynchronous revalidation. This implies that *stale content* is served while the new content is asynchronously revalidated.

The `Cache-Control` header has a couple of ways of expressing what should happen when an object expires, and how revalidation should happen.

Take this header for example:

```
Cache-Control: public, max-age=3600, stale-while-revalidate=300
```

This response can be stored in cache for an hour, but when it expires, the cache should serve the expired object for a maximum of 300 seconds past its expiration time, while backend revalidation takes place. As soon as the revalidation is finished, the content is fresh again.

> Varnish's *stale while revalidate* implementation is called *grace mode* and is covered later in this chapter.

Another revalidation mechanism is based on the `must-revalidate` keyword, as illustrated in the example below:

```
Cache-Control: public, max-age=3600, must-revalidate
```

In this case, the content is fresh for an hour, but because of `must-revalidate`, serving stale data is not allowed. This results in synchronous revalidation once the cached object has expired.

A third revalidation mechanism in `Cache-Control` is one that is a bit more aggressive:

```
Cache-Control: public, no-cache
```

Although `no-cache` was already discussed earlier as a mechanism to prohibit a response from being cached, its actual purpose is to force revalidation without explicit eviction.

It implies `must-revalidate`, but also immediately considers the object as stale.

How Varnish deals with Cache-Control

First things first: Varnish doesn't respect the `Cache-Control` as a request header, only as a response header.

Your web browser could send a `Cache-Control: no-cache` request header to avoid getting the cached version of a page.

One could argue that if Varnish truly wants to comply with HTTP's specs, it would respect this header, and not serve content from cache. But that would defy the entire purpose of having a reverse caching proxy, and this could result in a severe decline in performance and stability, not to mention an increased attack surface.

With that out of the way, let's look at which `Cache-Control` features Varnish *does* support by default:

- Varnish respects `s-maxage` and sets its *TTL* according to this value.
- Varnish respects `max-age` and sets its *TTL* according to this value, unless a `s-maxage` directive was found.
- Varnish respects the `private` directive and will not cache if it occurs.
- Varnish respects the `no-cache` directive and will not cache if it occurs.
- Varnish also respects the `no-store` directive, and will not cache when it occurs.

- Setting `max-age` or `s-maxage` to zero will cause Varnish not to cache the response.
- As mentioned, Varnish respects `stale-while-revalidate` and will set its *grace time* accordingly.
- There are two common `Cache-Control` directives that Varnish ignores:
- `public`
- `must-revalidate`

There is a `must-revalidate` Varnish implementation in the making, but since this would result in a breaking change, it can only be introduced in a new major version of Varnish. `must-revalidate` support in Varnish would result in *grace mode* being disabled.

3.2.3 Surrogates

The Edge Architecture Specification, which is a *W3C* standard, defines the use of *surrogates*. These *surrogates* are intermediary systems, that can act on behalf of the origin server.

These are basically *reverse proxies* like *Varnish*. Although some of them might be located close to the origin, others might be remote.

Varnish's typical use case in this context, is as a *local reverse caching proxy*. A typical example of *remote reverse caching proxies* is a *content delivery network (CDN)*.

Varnish is also *CDN software*. Although Varnish is primarily used in a *local* context, there are many use cases where Varnish is used in various geographical *points of presence*, to form a full-blown *CDN*.

In chapter 9, we'll discuss how Varnish can be used to build your own *CDN*.

Whereas a regular *proxy* only caches content coming from the origin, a *surrogate* can act on behalf of the origin and can perform logic *on the edge*. From offloading certain logic, to adding functionality *on the edge*, this makes *surrogates* a lot more powerful than regular *proxies*.

Surrogates can be controlled through specific *HTTP* headers:

- The `Surrogate-Capability` header is a request header, sent by the *surrogate* to announce its capabilities.
- The `Surrogate-Control` header is a response header, sent by the *origin*, to control the behavior of the *surrogate*, based on the capabilities it announced.

The Surrogate-Capability header

The `Surrogate-Capability` header is a request header that is not sent by the client, but by the *surrogate* itself. This header announces the *surrogate capabilities* that this reverse proxy has.

The origin that receives this header can act on these capabilities, and later control some of these *surrogate capabilities* through the `Surrogate-Control` header.

A `Surrogate-Capability` header is a collection of unique *device tokens*. Each one of these tokens relates to a specific *surrogate* that can be used to announce its own capabilities.

One of the most common *surrogate capabilities* is the capability to process *edge-side includes*.

> An *edge-side include* is a markup tag that is used to assemble content *on the edge*, using a *source attribute* that refers to an *HTTP endpoint*.
>
> When an origin server sends such an *ESI tag*, the *surrogate* will process the tag, call the endpoint, potentially cache that HTTP resource, and assemble the content as a single *HTTP* response.

Here's how a *surrogate* can announce *ESI* support:

```
Surrogate-Capability: varnish="ESI/1.0"
```

The Surrogate-Control header

Once a *surrogate* has announced its capabilities, the origin can control it using a list of directives in the `Surrogate-Control` header.

When we use our *ESI* example, this is how the origin would specify how the origin should process any *ESI* tags in the response:

```
Surrogate-Control: content="ESI/1.0"
```

And this is what an *ESI* tag looks like:

```
<esi:include src="http://example.com/header/" />
```

- The response body contains the *ESI* tag(s).
- The `Surrogate-Control` response header instructs the *surrogate* to process these tags as *ESI*. In chapter 4 we'll discuss *ESI* in more detail.

Surrogate caching

Although *surrogates* are about additional capabilities that go beyond basic *HTTP*, there is still a caching component to it. A `Surrogate-Control` header can contain directives like `no-store` and `max-age`, which are used to control the cacheability of a response.

Surrogates can use the `Surrogate-Control` header to set the cacheability of a response and its *TTL*. The requirement is that a `Surrogate/1.0` capability token is set in the `Surrogate-Capability` header, as illustrated below:

```
Surrogate-Capability: varnish="Surrogate/1.0"
```

When a *surrogate* announces `Surrogate/1.0` support, the `Surrogate-Control` caching directives have precedence over any *TTL* defined using the `Cache-Control` or `Expires` header.

Here's an example where we combine caching and *ESI* control:

```
Surrogate-Control: no-store, content="ESI/1.0"
```

Regardless of any `Cache-Control` header, the response will not be cached, but the output will be parsed as *ESI*.

It's also possible to indicate how long a *surrogate* should cache a response:

```
Surrogate-Control: max-age=3600
```

In the example above, a *surrogate* may cache this response for an hour. But it can get a bit more complicated when you look at the `max-age` syntax in the following example:

```
Surrogate-Control: max-age=3600+600
```

This `Surrogate-Control` example directs the *surrogate* to cache the response for an hour, but allows stale content to be served for another ten minutes, while revalidation happens.

> Although it's nice to have *revalidation* features within the `Surrogate-Control` syntax, it diverges from the conventional `stale-while-revalidate` syntax that is part of the `Cache-Control` header.

There's even an extra directive to control caching behavior, and that's the `no-store-remote` directive. `no-store-remote` will instruct *remote caches* not to store a response in cache, whereas *local caches* are allowed to store the response in cache.

> The implementation of `no-store-remote` is a bit arbitrary, and depends on whether or not a *surrogate* considers itself a *remote cache* or a *local cache*. It's up to the *surrogate* to decide, but generally, *surrogates* that are more than one or two hops from the origin server can call themselves *remote*. In most cases, *CDNs* fit that description.

Here's an example of `no-store-remote`:

```
Surrogate-Control: no-store-remote, max-age=3600
```

In this example, *local caches* with *surrogate capabilities* are allowed to cache the response for an hour, whereas *remote caches* aren't allowed to store this response in cache.

Surrogate targeting

The idea behind *surrogates* is that they can be deployed in various locations and can be part of a tiered architecture. When using a mixture of *CDNs* and local caches, various devices can have various capabilities.

Targeting specific *surrogate devices* is important when you want to control their individual capabilities. Each device emits its own *device keys* containing their individual capabilities.

Devices that are further along the chain may append capabilities to the `Surrogate-Capability` header as long as the *device key* remains unique.

Here's such an example:

```
Surrogate-Capability: varnish="Surrogate/1.0 ESI/1.0", cdn="Surro-
gate/1.0"
```

In this case, a device named `varnish` supports both the `Surrogate/1.0` specification and has *ESI* capabilities. There's also a device named `cdn` that only supports `Surrogate/1.0`.

These values were appended to the `Surrogate-Capability` header by the various *surrogates* in the content delivery chain and will be interpreted by the *origin*.

The *origin* can then issue the following `Surrogate-Control` header to control both devices:

```
Surrogate-Control: max-age=60, max-age=86400;varnish, max-
age=3600;cdn, content="ESI/1.0";varnish
```

Let's break this down:

- Any *surrogate device* that is not matched will store the response in cache for a minute *(`max-age=60`)*.
- The *surrogate device* named `varnish` will store the response in cache for a day *(`max-age=86400;varnish`)*.
- The *surrogate device* named `cdn` will store the response in cache for an hour *(`max-age=3600;cdn`)*.
- Additionally, the `varnish` *surrogate device* also has to process one or more *ESI* tags in this response.

Here's another combined example:

```
Surrogate-Control: max-age=3600, max-age=86400;varnish, no-store-re-
mote
```

And here's the breakdown:

- Any *surrogate device* that is not matched will store the response in cache for an hour (`max-age=3600`).
- The *surrogate device* named `varnish` will store the response in cache for a day (`max-age=86400;varnish`).
- Any *remote surrogate* will not be allowed to store this response in cache (`no-store-remote`).

Surrogate support in Varnish

Out-of-the-box, Varnish's support for *surrogates* is very limited. However, because capabilities and controlling features are so diverse, there is no *one-size-fits-all solution*. The *VCL* language is the perfect fit for the implementation of custom *edge logic*.

Varnish does respect the `Surrogate-Control: no-store` directive in its built-in behavior. Any other behavior should be declared using *VCL*.

In chapter 8, we'll be talking about decision-making *on the edge*, which is exactly the goal of *surrogates*.

3.2.4 TTL header precedence in Varnish

There's the `Expires` header, there's the `Cache-Control` header, and within `Cache-Control` there's `max-age` and `s-maxage`. Plenty of ways to set the *TTL*, but what is the order of precedence?

1. The `Cache-Control` header's `s-maxage` directive is checked.
2. When there's no `s-maxage`, Varnish will look for `max-age` to set its *TTL*.
3. When there's no `Cache-Control` header being returned, Varnish will use the `Expires` header to set its *TTL*.
4. When none of the above apply, Varnish will use the `default_ttl` runtime parameter as the *TTL* value. Its default value is *120 seconds*.
5. Only then will Varnish enter `vcl_backend_response`, letting you change the *TTL*.
6. Any *TTL* being set in *VCL* using `set beresp.ttl` will get the upper hand, regardless of any other value being set via response headers.

3.2.5 Cacheable request methods

You've probably heard the term *idempotence* before. It means applying an operation multiple times without changing the result.

In math, multiplying by zero has that effect. But in our case, we care about *idempotent request methods*.

An HTTP request method explicitly states the intent of a request:

- A `GET` request's purpose is to retrieve a resource.
- A `HEAD` request's purpose is to only retrieve the headers of a resource.
- A `POST` request's purpose is to add a new resource.
- A `PUT` request's purpose is to update a resource.
- A `PATCH` request's purpose is to partially update a resource.
- A `DELETE` request's purpose is to remove a resource.

This should sound quite familiar if you've ever worked with *RESTful APIs*.

The only idempotent request methods in this list are `GET` and `HEAD` because executing them does not inherently change the resource.

This is not the case with `POST`, `PUT`, `PATCH`, and `DELETE`.

That's why Varnish only serves objects from cache when they are requested via `GET` or `HEAD`.

> Caching *non-idempotent* requests is possible in Varnish, but it's not conventional behavior. Using custom *VCL* code and some *VMODs*, it can be done. But it depends heavily on your use case. See *chapter 8* for a section about *caching POST requests*.
>
> Note: because of HTTP's flexibility, you can of course design idempotent `POST` requests and non-idempotent `GET` ones, but the REST approach is the overwhelming norm.

3.3.6 Cacheable status codes

As described in the previous section: when receiving a *client request*, a *reverse caching proxy* should be picky as to what request methods it deems cacheable.

The same thing applies for *backend responses*: only backend responses containing certain status codes are deemed cacheable. These are all defined in section 6.1 of RFC 7231.

Varnish only caches responses that have the following status code:

- `200` OK
- `203` Non-Authoritative Information
- `204` No Content
- `300` Multiple Choices
- `301` Moved Permanently
- `302` Moved Temporarily
- `307` Temporary Redirect
- `304` Not Modified
- `404` Not Found
- `410` Gone
- `414` Request-URI Too Large

Responses containing any other status code will not be cached by default.

3.2.7 Cache variations

Throughout this chapter, we talk about HTTP, and more specifically in this section, about the caching aspect of it. Through a variety of headers, we can instruct Varnish what to cache and for how long.

But there's another instruction we can assign to a cache: *how* to store the object. A cached object should be retrieved through its unique identifier. From an HTTP perspective, each resource already has a conventional way to be identified: *the URL*.

HTTP caches like Varnish will use the URL as the hash key to identify an object in cache.

If the URL is the unique identifier, but the content differs per user, it seems as though you're in trouble, and the response will not be cached. But that's not really the case because HTTP has a mechanism to create cache variations.

Cache variations use a secondary key to identify variations of the object in cache.

It's basically a way for the origin to add information to the hash key to complement the cache's initial hashing.

The vary header

The way HTTP requests a cache variation is through the `Vary` response header. The value of this `Vary` header should be a valid request header.

For each value of the *request header* that is used in the `Vary` response header , a secondary key will be created to store the variation.

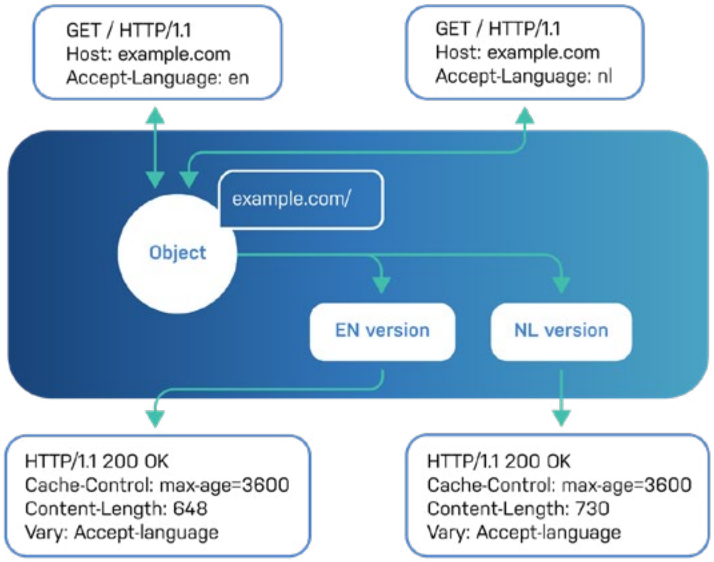
Accept-Language variation example

A very common example: *language detection*.

Although in most cases, *splash pages* with a language selection option are used for multilingual websites, HTTP does provide a mechanism to automatically detect the language of the client.

Web browsers will expose an `Accept-Language` header containing the language the user prefers. When your website, or API, detects this, it can automatically produce multilingual content, or automatically redirect to a language-specific page.

But without a cache variation, the cache is unaware of this multilingual requirement and would store the first occurrence of the page. This will result in a language mismatch for parts of the audience.



```
GET / HTTP/1.1
Host: example.com
Accept-Language: en
```

```
GET / HTTP/1.1
Host: example.com
Accept-Language: nl
```

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Content-Length: 648
Vary: Accept-language
```

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Content-Length: 730
Vary: Accept-language
```

Cache variations

By issuing `Vary: Accept-Language`, Varnish is aware of the variation and will create a separate secondary key for each value the `Accept-Language` may have.

> Disclaimer: this is an oversimplified example. In reality there are more things to consider before creating an `Accept-Language` cache variation, which will be covered in the next section.

One thing to note, which will become important for cache invalidation: variants actually share the hash key, so they can be invalidated in one go.

Hit-rate considerations

When dealing with personalized content, you try to cache as much as possible. It may be tempting to jam in cache variations wherever you can.

However, it is important to consider the potential hit rate of each variation.

Take for example the following request:

```
GET / HTTP/1.1
Host: example.com
Cookie: language=en
```

The *language cookie* was set, which will be used to present multilingual content.

You could then create the following cache variation:

```
Vary: Cookie
```

There are so many risks involved, unless you properly sanitize the user input.

Problem number one: the user can change the value of the cookie and deliberately, or even involuntarily, increase the number of variations in the cache. This can have a significant impact on the hit rate.

In reality there will probably more cookies than just this *language cookie*. An average website has numerous tracking cookies for analytics purposes, and the value of some of these cookies can change upon every request.

This means every request would create a new variation. This wouldn't just kill the hit rate, but it would also fill up the cache to a point that Varnish's *LRU eviction mechanism* would forcefully have to evict objects from cache in order to free up space.

That's why it's imperative to *sanitize user input* in order to prevent unwarranted variations.

Because this is what a browser could send in terms of `Accept` and `Accept-Language` headers:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Language: nl-NL,nl;q=0.9,en-US;q=0.8,en;q=0.7
```

It's very tricky to create variations based on either of those headers because the secondary key that will be created is the literal string value of the varied header.

Without proper sanitization, this will create too many variations, which will drive down your hit rate.

Sanitizing user input

The solution is to clean up the user input using *VCL*.

> We know we're getting ahead of ourselves here, because *VCL* will be properly covered in the next chapter. However, it is interesting to know, from a practical point of view, how to properly sanitize user input for these *accept headers*.

Here's some example *VCL* that leverages `vmod_accept`:

```
vcl 4.1;

import accept;

sub vcl_init {
    new lang = accept.rule("en");
    lang.add("nl");
}

sub vcl_recv {
    set req.http.Accept-Language = lang.filter(req.http.Accept-Lan-
guage);
}
```

`vmod_accept` will simplify the values of `Accept-Language` to make variations more controllable. It does this based on a *whitelist*.

The whitelist is named `lang` and lists the allowed values for the `Accept-Language` header.

By executing the `lang.filter()` function, `vmod_accept` will look the input for the `Accept-Language` header, sent by the browser, and will keep the first match. If no match is found, the filter will take the default value.

In this case the allowed values are `en` and `nl`. If either of these languages is found in the `Accept-Language` header, one of them will be selected, based on the first occurrence. If none of them are found, the default language will become `en`.

This is what comes in:

```
Accept-Language: nl-BE;q=0.9,en-US;q=0.8,en;q=0.7
```

And because the first match is `nl-BE`, `vmod_accept` will turn this into:

```
Accept-Language: nl
```

You can then safely return a `Vary: Accept-Language`, knowing that only two variations will be allowed.

This can also be done for other headers, using other modules or VCL constructs.

Varying on custom headers

Unfortunately, it's not always possible to sanitize the headers you want to vary on. Cookies are a perfect example: you cannot sanitize certain parts for the sake of cache variations. You risk losing valuable data.

A potential solution is to *create a custom request header* that contains the value you want to vary on.

Imagine the following cookie header:

```
Cookie: sessionid=615668E0-FC89-4A82-B7C1-0822E4BE3F87, lang=nl, ac-
cepted_cookie_policy=1
```

You want to create a cache variation on the value of the *language cookie*, but you don't want to lose the other cookies in the process.

What you could do is create a custom `X-Language` header that contains the value of the *language cookie*. You could then perform the following cache variation:

```
Vary: X-Language
```

Here's the *VCL* code to achieve this:

```
sub vcl_recv {
    set req.http.x-language = regsub(req.http.cookie,"^.*la
ng=([^;]*);*.*$","\1");
    if(req.http.x-language !~ "^en|nl$") {
        set req.http.x-language = "en";
    }
}
```

The *VCL* code has a similar effect to the `Accept-Language` sanitization example:

- Only the `en` and `nl` values are allowed
- If another value is used, we default back to `en`

As a developer, you can then opt to vary on `X-Language`.

This custom variation can also be processed using *VCL* only, without the need for an explicit cache variation in HTTP.

Here's the *VCL* code:

```
sub vcl_recv {
    set req.http.x-language = regsub(req.http.cookie,"^.*la
ng=([^;]*);*.*$","\1");
    if(req.http.x-language !~ "^en|nl$") {
        set req.http.x-language = "en";
    }
}

sub vcl_backend_response {
    set beresp.http.vary = beresp.http.vary + ", x-language";
}
```

At the end of `vcl_backend_response`, Varnish will check the `vary` header and create the secondary key, as if the header had been provided by the origin directly.

Although this makes life easier for developers, it's not a portable solution. We always prefer using HTTP as much as possible and then resort to *VCL* when HTTP cannot solve the problem.

3.3 Varnish built-in VCL behavior

In the previous section, we talked about how Varnish deals with the `Cache-Control` header. It talked about how Varnish deals with the different values internally.

Parts of this behavior are implemented in the Varnish core, but other parts are implemented in the so-called *built-in VCL*.

The *built-in VCL* contains a set of rules that will be executed by default, even if they are not specified in your own *VCL file*. It is possible to bypass the *built-in VCL* through a `return` statement, but this chapter assumes that this does not happen. The *built-in VCL* provides much of the *safe-by-default* behavior of *Varnish*, so be very careful if you decide to skip it.

The *VCL language* is tightly connected with the *Varnish finite state machine*, where the various VCL subroutines correspond to states in the machine. Since the *built-in VCL* executes last, each *VCL subroutine* ends with a `return` statement, which controls the flow of the machine. Fully understanding the finite state machine requires an understanding of the *built-in VCL*, and vice versa. Luckily, a basic understanding of both is sufficient to solve most use cases for *VCL*.

We won't be going over the *built-in VCL code* in this section, only the *built-in VCL behavior*. In the next chapter we'll cover *VCL* in detail, including the syntax. The corresponding *built-in VCL code* will be presented in the next chapter, which will make a lot more sense.

It is important to note that the *built-in VCL* is very cautious in its implementation. This prevents *Varnish* from caching anything that shouldn't be cached, but this can result in a very low hit rate when the backend does not provide good caching headers or uses cookies. How to mitigate this is explained in chapter 4.

Let's look at the concrete behavior that the built-in VCL implements.

3.3.1 When is a request cacheable?

The first task for the *built-in VCL* is to decide if a *request* is cacheable.

When Varnish receives an HTTP request from the client, it first looks at the *HTTP request method* to decide what needs to happen.

Cacheable request methods

In the previous section, we talked about cacheable request methods. This logic is also part of the *built-in VCL*.

If it turns out the *request method* equals `GET` or `HEAD`, Varnish will allow the request to be served from cache. Other request methods will immediately result in a *pass* or *pipe* to the backend.

> A small side note about `GET` requests: although it's not that common, a `GET` request can contain payload in its request body. However, Varnish will strip the request body from a `GET` request before sending it to the backend.

Invalid request methods

The *cacheability* of a request method isn't the only thing Varnish cares about. Varnish will only deal with *request methods* it can handle.

If the *request method* is `PRI`, which shouldn't normally happen, Varnish will stop execution with an `HTTP 405 Method not allowed` error. That's because `PRI` is part of `HTTP/2`, and is handled in the Varnish core when `HTTP/2` is enabled.

If you receive a request with such a method, it means `HTTP/2` wasn't properly configured, and you're receiving an `HTTP/2` request on an `HTTP/1.1` server.

By default Varnish will only handle the following request methods:

- GET
- HEAD
- POST
- PUT
- PATCH
- DELETE
- OPTIONS
- TRACE

For any other *request method*, other than `PRI` of course, the *built-in VCL* will turn the connection into a *pipe*.

This means that Varnish doesn't only pass the request to the backend, but it no longer treats the incoming request as an HTTP request.

Varnish opens up a *pipe* to the backend, and just sends the incoming request through as regular *TCP*, without adding any notion of *HTTP*.

Please note that there is no *pipe* in `HTTP/2`. For `HTTP/1.1`, websockets is the main use case.

State getting in the way

As mentioned before, caching when *state* is involved is tricky.

Stateful data is usually personalized data. Storing this kind of data in the cache doesn't make sense. Previously we mentioned *cache variations* as a potential solution.

But by default, Varnish doesn't serve any content from cache that either contains a `Cookie` request header, or an `Authorization` request header.

> In the real world, cookies will almost always be used. That's why you'll need to write custom *VCL* for your specific application, which caches certain cookies, and strips off others. In the next chapter, we'll show you how to do this.

3.3.2 How does Varnish identify objects in cache?

Once Varnish decides that an incoming request is cacheable, it needs to look up the corresponding object in cache.

When an object is stored in cache, a *hash key* is used to identify the object in cache. As previously explained, the *hostname* and the *URL* are used as identifiers to create this hash.

The *hostname* is the value that comes out of the `Host` request header. When the request doesn't contain a `Host` header, Varnish will use the *its own server IP address* instead. This can only happen when a request uses an old version of HTTP.

As of `HTTP/1.1`, a `Host` header is no longer optional. Varnish will return an *HTTP 400* error when it notices an `HTTP/1.1` request without a `Host` header.

3.3.3 Dealing with stale content

One of the core caching principles is the *time to live (TTL)*. This is not part of the *built-in VCL*, but part of the core caching logic of Varnish.

We talked about `Cache-Control` values, and about the `Expires` header. We even talked briefly about overriding the *TTL* in *VCL code*.

Varnish will use these mechanisms to come up with the *TTL* of an object. And as long as the *TTL* is greater than zero, the object is deemed *fresh*, and will be served from cache.

Varnish will check the freshness of an object upon every *cache hit*. If it turns out the *TTL* has become zero or less, the object is deemed *stale* and revalidation needs to happen.

Because of the so-called *grace mode*, Varnish is able to serve stale data to the client while asynchronously revalidating the content with the backend server. We already talked about this when explaining `stale-while-revalidate` behavior.

Varnish will check the sum of the *TTL* and the *grace value*, and if it is greater than zero, it will still serve the stale data while performing a *background fetch*.

If the sum is zero or less than zero, the content will cause a *cache miss*, and a *synchronous fetch* will happen.

The `default_grace` runtime parameter defines the *default grace period* when the grace period is not specified in a response header from the backend. The default value for `default_grace` is *ten seconds*, but it can be changed in the command line or dynamically when Varnish is running. If an object's grace is ten seconds on insertion, then the object will be asynchronously revalidated until ten seconds after the *TTL* has expired. When the *grace period* has passed, a request to this object will be a *cache miss* and will trigger a *synchronous fetch* to the origin.

The grace period can also be set in *VCL* by assigning a value to the `beresp.grace` variable, but this will be discussed in the next chapter. As mentioned before: the `Cache-Control` header also has the `stale-while-revalidate` directive to set the grace period.

3.3.4 When does Varnish store a response in cache?

There's a difference between a *cacheable* request, and deciding that a response should be *stored in cache*.

The former decision is made when Varnish receives an incoming request. Chances are that the object is already stored in cache, which results in a *hit*. If that's not the case, it's a *miss*.

When a request is not served from cache, a *backend request* is made. When the backend responds, Varnish will receive the *HTTP response* and will decide what to do with it.

The decision-making process as to whether or not to cache is based on *response headers* sent by the backend.

The first step in the decision-making process is interpreting the *TTL*. If the *TTL*, coming from `Cache-Control` or `Expires` headers is zero or less, Varnish will decide not to store this response in cache.

In the next step, the *built-in VCL* will check if there's a `Set-Cookie` in the response. The `Set-Cookie` header is used to store state in the client, and is usually used for private information or as a unique identifier for the user. For this reason the *built-in VCL* will mark the response as *uncacheable*, so that no other clients can get the same, potentially private, `Set-Cookie` header.

In the previous section we talked about *surrogates*. Varnish checks whether or not a `Surrogate-Control: no-store` is set. If this is the case, Varnish will not store the response in cache.

When there's no `Surrogate-Control` header being returned, Varnish will also look at the semantics of the `Cache-Control` header, beyond the interpretation of the *TTL*. If terms like `no-cache`, `no-store`, or `private` occur in this header, Varnish will decide not to store this response in cache.

And finally, if a `Vary: *` header is sent by the origin, Varnish won't cache the response either. Because varying on all headers makes no sense if you want to cache a response.

3.3.5 What happens if the response couldn't be stored in cache?

When Varnish decides that a response is not cacheable, for the reasons mentioned above, the response is directly served to the client that requested it without being cached.

However, Varnish will store some metadata in the cache for *uncacheable resources*. Varnish will create a *hit-for-miss* object to recognize the fact that the response was *uncacheable*.

The purpose of this *hit-for-miss* object is to bypass the waiting list for future requests to this resource.

Requests for cacheable resources can be put on the waiting list while *request coalescing* happens.

Request coalescing was explained in *chapter 1*. It makes sure that multiple requests that can be satisfied by a single backend request are not sent to the backend, but instead are put on a waiting list. When the backend response is received, the response is sent to all satisfiable requests that were queued in the waiting list.

By marking a request for a specific resource *uncacheable* through a *hit-for-miss* object, we avoid the waiting list and immediately issue a backend request. We do this knowing that a request for this resource will probably not be satisfied through *request coalescing*.

As a consequence, we avoid potential *request serialization*. This term refers to requests being processed *in a serial manner* instead of in parallel. Request serialization causes extra latency and even becomes a bottleneck when there are enough clients requesting the same URL.

Request coalescing is otherwise a powerful Varnish feature, and the waiting list is part of this implementation. But for uncacheable content, the waiting list would become counterproductive. That's why the *hit-for-miss* is there to deliberately bypass it.

The *hit-for-miss* logic is actually quite forgiving: *hit-for-miss* objects can be replaced with actual cached responses when the next backend response is considered cacheable.

A *hit-for-miss* object is kept in cache for a certain amount of time. By default a *hit-for-miss* object has a *TTL* of two minutes. This *TTL* represents the upper limit, but if the next response is cacheable, the *hit-for-miss* object is replaced with the actual cacheable response.

3.4 Range requests

As mentioned in the first section of this chapter: *HTTP* is the go-to protocol these days. It's true for all kinds of implementations, which are surely beyond the scope of the initial *HTTP/1.1* use cases.

The speed of internet connections has gone up, and greater bandwidth allows for larger data transfers. At this point in time, about 80% of the internet's bandwidth is consumed by online video.

In *chapter 10* we'll be talking about *OTT video streaming*, and how Varnish can be used to accelerate these streams.

It's realistic that you'll see output like this, where the size of the `Content-Length` response header is massive:

```
HTTP/1.1 200 OK
Content-Length: 354648464
```

This is a *338 MB* response. For *OTT video streaming*, you'll want to chop this up into several smaller files to improve the user experience, to improve your bandwidth consumption, and to reduce the strain on your backend systems.

HTTP already has a built-in mechanism to serve partial content. This mechanism is called *byte serving* and allows clients to perform *range requests* on resources that support serving partial content.

Another use case for *range request* is a download manager that can *pause and resume* downloads. Maybe you want to download the first *300 MB* right now, and continue downloading the remaining *38 MB* tomorrow. That's perfectly realistic using *range requests*.

3.4.1 Accept-Ranges response header

A web server can advertise whether or not it supports *range requests* by returning the `Accept-Ranges` response header.

When *range request* support is active on the web server, the following header can be returned:

```
Accept-Ranges: bytes
```

This means the range unit is expressed in *bytes*.

The value could also be *none* in case the server actively advertises it doesn't support *range requests*:

```
Accept-Ranges: none
```

Based on either of these values, a client can decide whether or not to send a *range request*.

3.4.2 Range request header

A client can send a `Range` request header and let the server know which range of bytes it wishes to receive.

Here's an example:

```
Range: bytes=0-701615
```

This example fetches a byte range starting at the beginning up to and including the *701615-th* byte.

Here's another example:

```
Range: bytes=701616-7141367
```

Whereas the previous range ended at the *701615-th* byte, this example picks up from the *701616-th* byte until the *7141367-th* byte.

When you perform a *range request*, you're not requesting the full response body. The server acknowledges the fact that you're receiving *partial content* through the `HTTP 206 Partial Content` status code.

3.4.3 Content-Range response header

Whereas an `Accept-Ranges` response header is returned regardless of the type of request, a `Content-Range` response header is only sent when an actual *range request* occurs.

Here's an example of a `Content-Range` header that is based on the `Range: bytes=701616-7141367` range request:

```
Content-Range: bytes 701616-7141367/354648464
```

The header matches the range that was requested, but it also contains the *total byte size* of the resource.

If we do the math, we can come to two conclusions:

- `7141367 - 701616 = 6439752`, which corresponds to the value of the `Content-Length` header we also receive from the server
- `354648464` that is part of the `Content-Range` value matches the value of the `Content-Length` header if you performed a regular request on this resource

Basically, the `Content-Range` header confirms the range you're receiving, and the upper range limit you can request.

3.4.4 What if the range request fails?

The consequences of a *range request failure* depend on the implementation. But if the web server failed to deliver the requested range, it will return an `HTTP 416 Range Not Satisfiable` error.

But that is typically used when your range request goes *out of bounds*. It is also possible that the client is performing a *range request* on a web server that doesn't support this.

In that case, there various outcomes:

- You could receive an `HTTP 406 Not acceptable` error.
- You could receive any other *400-style* HTTP error.
- You could receive regular *HTTP 200* output, containing the full payload if the server ignores your range request, which is usually what happens.
- But if you want to properly perform range requests, and avoid failures, it's a bit of a *chicken or egg* situation:
- Do you first perform a check to see if the server returns an `Accept-Ranges: bytes` header?
- Or do you just send a *range request* and deal with the consequences?

You could send a `HEAD` request first, instead of a `GET` request, and look for an `Accept-Ranges` header before performing the actual *range request*.

3.4.5 Range request support in Varnish

Varnish supports *client-side range requests*. This means that if a client sends a `Range` header that Varnish can satisfy, the client will receive the requested range, whether the origin supports it or not.

However, Varnish disables *backend range requests* for cached requests by default. This means that if a *range request* for a resource results in a *cache miss*, a regular HTTP request is sent to the origin. Varnish will store the full response in cache and will return the requested range to the client.

In most cases, this allows efficient collapsing of the requests, but for large objects, it can lack efficiency.

Impact on the origin

If the requested resource is a really large file, Varnish will need to ingest and cache the beginning of the file before it reaches the requested range that it will then serve.

If network throughput between Varnish and the origin is poor, the client will experience additional latency. Also, if returning this resource consumes a lot of server resources at the origin, it will also add latency.

However, if the requested range for this cache miss starts at the first byte, Varnish will leverage *content streaming*. This means that the client will not have to wait until the complete resource is stored in Varnish and will receive the data in chunks, as it is received by Varnish.

If you don't want to support ranges, you can just disable the `http_range_support` runtime parameter.

Backend range requests using VCL

Although Varnish doesn't natively support *backend range requests*, we can write some *VCL* to get the job done.

As mentioned before, *VCL* will be covered in detail in the next chapter. We'll just explain the concept of the example below, without focusing too much on syntax:

```
sub vcl_recv {
    # if there's no Range header we like, use
    # "0-" which means "from the 0-th byte till the end"
    if (req.http.Range ~ "bytes=") {
        set req.http.x-range = "req.http.Range";
    } else {
        set req.http.x-range = "bytes=0-";
    }
}

sub vcl_hash {
    hash_data(req.http.x-range);
}

sub vcl_backend_fetch {
    set bereq.http.Range = bereq.http.x-range;
}

sub vcl_backend_response {
    if (beresp.status == 206) {
        set beresp.ttl = 10m;
        set beresp.http.x-content-range = beresp.http.Content-Range;
    }
}

sub vcl_deliver {
    if (resp.http.x-content-range) {
        set resp.http.Content-Range = resp.http.x-content-range;
        unset resp.http.x-content-range;
    }
}
```

When Varnish receives a `Range` request header, it will store its value in a custom `x-range` request header. This value will also be added to the *hash key* to create a new cached entry per range.

Because Varnish will remove the `Range` header before initiating a backend fetch, we will explicitly set a new `Range` header containing the value of `x-range`.

When the backend responds with a `Content-Range` header and an *HTTP 206* status code, we'll store the value of the `Content-Range` header in a custom `x-range` header, knowing that Varnish will strip off the original `Content-Range` header.

Before returning the range to the client, we set the `Content-Range` response header with the value that was captured from the origin.

At some point in time, Varnish will support *native backend range requests* and will probably store ranges separately in the cache. There might even be a more efficient solution than the VCL example above. But until then, you need to be aware of the limitations, potentially use the *VCL* example, and plan accordingly.

3.5 Conditional requests

Caching can alleviate a lot of stress from your origin servers, and as a consequence your overall stability increases and latency gets reduced.

But caches don't fill themselves up: content needs to be fetched from the origin. Depending on the hit rate of your cache, this can still result in heavy load on the origin, increased latency, and an overall decrease in stability.

Even when revalidating stale content, or when performing range requests, the full result needs to be fetched. This can be resource intensive on the origin side.

But if you optimize your origin for *conditional requests*, origin fetches will become a lot more efficient.

3.5.1 304 Not Modified

As this chapter is all about HTTP, it should come as no surprise to learn that HTTP has built-in support for *conditional requests*.

The idea is that you present the origin a *fingerprint* of stale content that you want to revalidate. When the corresponding fingerprint sent by the client matches the current fingerprint of the content, the origin can return an `HTTP 304 Not Modified` status.

An *HTTP 304* response doesn't require a body to be sent, as it implies that whatever the client has stored in cache is the most recent version of the content.

If the fingerprint doesn't match, a regular `HTTP 200 OK` is returned instead.

3.5.2 Etag: the fingerprint

The *fingerprint* referred to is an arbitrary value that is set by the origin, which is returned through an `Etag` response header.

There is no conventional format for this header; the only requirement is that it is unique for the content that is returned.

Most web servers can automatically generate an `Etag` for resources that are files on disk. For web applications that use *URL rewriting*, it is up to the application itself to generate the `Etag`.

If you use an *MVC framework*, generating an `Etag` is quite simple. You could take a hash of the content before returning it, and use this hash as the value of the `Etag` response header. You can use algorithms like `md5` or `sha256` to create the hash.

Here's an example of an HTTP response containing an `Etag`:

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600, s-maxage=86400
Etag: "5985cb907f843bb60f776d385eea6c82"
```

3.5.3 If-None-Match

When a client receives an HTTP response that contains an `Etag` header, it can keep track of the `Etag` value, and send it back to the server via an `If-None-Match` header.

Based on the value of the `If-None-Match` header, the server can compare this value to the `Etag` value it was about to send. If both values match, the client has the most recent version of the content. The request can be satisfied by an `HTTP 304 Not Modified` response.

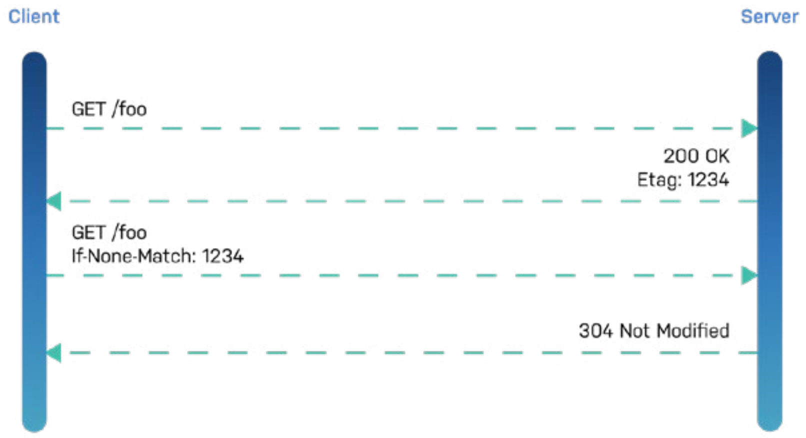
If the values don't match, an `HTTP 200 OK` will be returned, containing the full payload.

The `If-None-Match` request header is automatically added to requests by typical web browsers. When performing *command-line HTTP requests* using `curl` for example, the `If-None-Match` header should be added manually.

3.5.4 The workflow

On the one hand you have an `Etag` response header, on the other hand, you have an `If-None-Match` request header, and somehow the `HTTP 304 Not Modified` fits into the story as well.

Here's the workflow that will help you make sense of it all:



Conditional request workflow

7. The *client* sends a first request to the server for `/foo`.
8. The *server* replies with an `HTTP 200 OK`.
9. The *server* attaches an `Etag: 1234` header.
10. The *client* keeps track of the `1234` value.
11. The *client* sends another request to the server for `/foo`.
12. The *client* attaches an `If-None-Match: 1234` request header to that request.
13. The *server* recognizes the fact that `If-None-Match: 1234` was set by the client.
14. The *server* matches the `1234` value to whatever the `Etag` is supposed to be for this response.
15. The *server* notices that `1234` is still the *up-to-date* fingerprint of the content.
16. The *server* sends an `HTTP 304 Not Modified` to the client without any payload.
17. The *client* recognizes that the content hasn't been modified, and keeps serving whatever is stored in its cache.

3.5.5 Strong vs weak validation

An `Etag` is a specific *validator* in HTTP. So-called *strong validation* implies that the content that is represented by this *validator* is *byte-for-byte* identical.

Weak validation implies that the response is not *byte-for-byte* identical to the version it is comparing itself to, yet the content can be considered the same. For example, the uncompressed and compressed versions of an object.

A weakened `Etag` is prefixed with a `W/`. Here's an example:

```
Etag: "W/5985cb907f843bb60f776d385eea6c82"
```

This means that if an `If-None-Match` request header is received containing this value, the server should know how to validate the content, knowing it will not be *byte-for-byte* identical.

A practical use case is when the main content of a page remains the same, but certain ads, and certain information in the footer, might differ.

> Weak validation can get quite complicated. It requires the validating system to be aware of the subtleties of content, and it must be able to spot content that is not modified, even if the payload differs.

Varnish also emits *weakened Etags*, when the requested *content encoding* differs from what was stored in cache. We'll talk about this in just a minute.

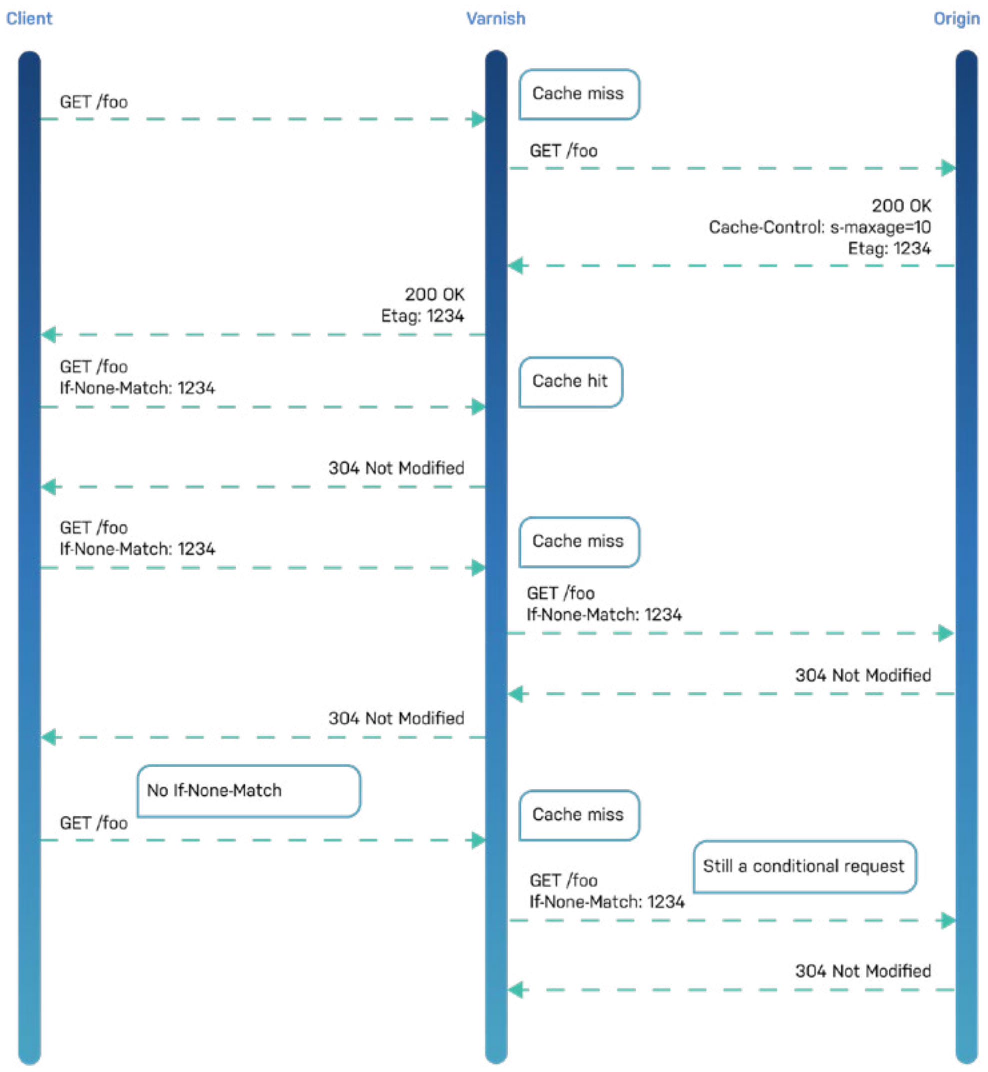
3.5.6 Conditional request support in Varnish

Varnish supports *conditional requests* in both directions. This means Varnish will return an `HTTP 304 Not Modified` when a client sends a matching `If-None-Match` header.

But it also means that Varnish will send an `If-None-Match` header to the origin on certain *cache misses*, hoping to receive an `HTTP 304 Not Modified`.

Conditional request workflow in Varnish

Here's a diagram that illustrates the workflow within Varnish:



Conditional request workflow in Varnish

Let's walk through it:

18. When a *client* requests a resource for the first time, it will be a *cache miss.*
19. *Varnish* will fetch the content from the origin.
20. The *origin* returns the content, and adds a `Cache-Control: s-maxage=10` header to indicate that the content should be cached for ten seconds.
21. The *origin* also includes an `Etag: 1234` header to announce the fingerprint of the resource.

22. *Varnish* stores the object in cache for ten seconds and returns the response, including the *Etag*.
23. The *client* receives the response from Varnish, and keeps track of the *Etag*.
24. The *client* sends a new request to Varnish within ten seconds and adds `If-None-Match: 1234`.
25. *Varnish* can deliver the object directly from cache because it is a *hit*, and the content is *fresh*.
26. Because the *Etag* matches, *Varnish* will not return a response body and will use the `HTTP 304 Not Modified` status code to indicate that the client has the most recent version of the object.
27. Sometime later, the *client* does another request for the same resources with the same `If-None-Match` header.
28. *Varnish* finds the object in cache, but it has expired, so it results in a *cache miss*.
29. *Varnish* will send a backend request to the origin, including the `If-None-Match: 1234` header.
30. The *origin* notices this header coming from Varnish, matches it to the *Etag*, and returns a bodyless *HTTP 304 Not Modified* response.
31. *Varnish* knows it still has the most recent version of the object in cache and can safely return a *HTTP 304 Not Modified* to the client.
32. For some reason the *client* no longer sends the `If-None-Match` header to Varnish for its next request.
33. *Varnish* finds the object in cache, but notices it has expired.
34. Because *Varnish* still has the *Etag* stored internally, it will pass it to the origin using the `If-None-Match: 1234` request header.
35. The *origin* acknowledges that this is still the latest version of the content and responds with an `HTTP 304 Not Modified` response.

Varnish supports *conditional requests* at the *client level* and at the *backend level*. But what is even more interesting is that once Varnish stores an *Etag*, it can use it for conditional requests to the backend for client requests that didn't contain an `If-None-Match` header.

Grace vs keep

When the *TTL* of an object is still greater than zero, the content is still fresh, and it can be served from cache.

As we've seen in a previous section, expired objects that still have some *grace* left can be revalidated asynchronously. This revalidation can also be done *conditionally*.

If the *grace period* hasn't expired, and Varnish has an *Etag* for this object, Varnish will send a background fetch to the origin, including the `If-None-Match` header. If the *Etag* matches, the origin will reply with an `HTTP 304 Not Modified` status code.

All of this happens in the background, while incoming client requests receive the stale object.

When the *grace period* has expired, the *keep period* kicks in, and revalidations to the origin become synchronous, meaning that clients will have to wait until the revalidation is finished.

But when that keep time has expired, the revalidation is unconditional, meaning that the response is supposed to be a regular `HTTP 200 OK`. This makes sense since after the *keep period* has expired, the object isn't in cache any more, so it needs to be fetched fully again.

Here's an overview to summarize *TTL*, *grace*, and *keep*:

```
TTL > 0 == fresh
TTL + grace > 0 == stale, (condtional) revalidation with background
fetch possible
TTL + grace + keep > 0 == stale, conditional synchronous revalidation
possible
TTL + grace + keep <= 0 == stale, unconditional synchronous revalida-
tion
```

3.5.7 Optimizing the origin for conditional requests

If you optimize your web application for *conditional requests*, you can take away a lot of stress from the origin system.

Some context

The fact that a `HTTP 304 Not Modified` has no response body reduces the size of the response on the wire. The first observation is that *conditional requests* are good for your bandwidth.

But in a web acceleration context, the real problems are increased CPU usage, running out of memory, and *disk I/O*. When a web application is under heavy load, latency can severely increase, and at a certain point the application can become non-responsive, which affects stability.

The point of running Varnish is to avoid latency and stability issues, but even with a properly configured Varnish server, there can still be plenty of traffic to the origin system:

- You can have a low *hit rate* because of diverse traffic patterns hitting *non-cached resources*.
- You can have a low *hit rate* because of *low TTLs*.

In each of these situations, there will be more traffic on the origin, which can increase the load on that system.

Exit early

The crux of *conditional requests* is to send an `HTTP 304 Not Modified` as early as possible. In order to take advantage of this in your web application, you need your application framework to validate the *Etag* as quickly as possible and exit as early as possible.

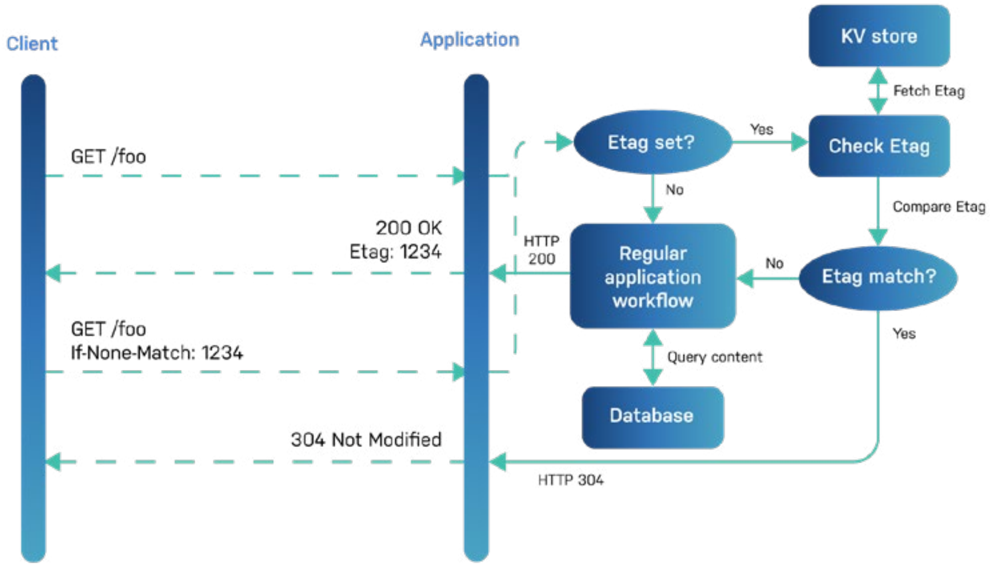
This means that you should have quick access to the *Etag* without having to go through your entire application logic.

When content is created or edited, your application should store the *Etag* in a *key-value store* or *database* that has very quick read access. When your application logic has to validate the *Etag*, a very *low overhead* call is made to the *Etag* storage, the `If-None-Match` header is matched to the *Etag*, and if they match, an `HTTP 304 Not Modified` is returned immediately.

This so-called *key-value store* can either be the local memory of the application server, or well-known products like *Redis* or *Memcached*. As long as read access is fast, and the overhead for retrieval is low, you've got yourself a good solution.

When properly optimized, the application will exit correctly, without a full bootstrap of the framework being required, and without access to typical database systems. This will result in very quick response times, and a very low-resource footprint on the server.

When the *Etag* doesn't match, the regular application flow takes place, resulting in an `HTTP 200 OK` response, and no real performance gain.



Conditional requests - application workflow

As you can see in the image above, there is a separate component that takes care of the *Etag check*. This component can be part of your regular application code, under the form of a *pre-dispatch hook* in your *MVC framework*.

It could also be a separate service, if need be.

Leveraging Varnish

As mentioned, Varnish supports *conditional requests* both on the *client side* and the *back-end side*. But if you know how to optimize your web application for *conditional requests*, you can leverage *Varnish* to get even better results.

You could *lower your TTLs*, without the risk of destabilizing your origin with increased requests. For HTTP resources that are updated on a frequent basis, you could even set the *TTL* to one second, and still have a stable origin:

- *Grace mode* will ensure asynchronous revalidation happens, while clients receive stale data.
- *Request coalescing* will ensure that only one request per URL is sent to the origin server.

Varnish has *purging* and *banning* capabilities to remove specific objects from the cache, which we'll cover in *chapter 6*. But by leveraging *conditional requests* in conjunction with low *TTLs*, there would be no need to actively remove content from the cache because the cache lifetime could just be a couple of seconds.

3.5.8 Last-Modified and If-Modified-Since as your backup plan

Etags aren't the only way to perform conditional requests. There's also the `Last-Modified` header to indicate when the content was last changed.

Here's an example:

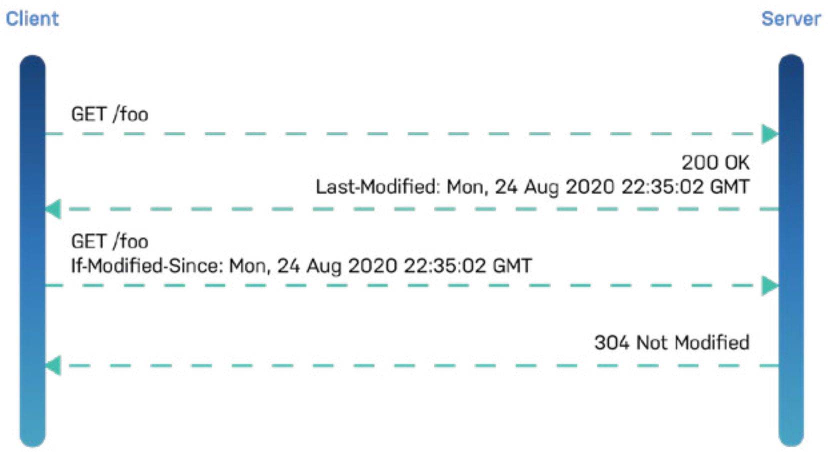
```
HTTP/1.1 200 OK
Cache-Control: max-age=3600, s-maxage=86400
Last-Modified: Mon, 24 Aug 2020 22:35:02 GMT
```

The origin indicates that the response is cacheable and can be cached by Varnish for a day, and by the browser for an hour. But the response also indicates that the content was last modified on *Monday August 24th at 22:35:02 GMT.*

The `Last-Modified` value can be stored by the client and will be sent to the server in the form of the `If-Modified-Since` header.

If the content wasn't modified since the value of the `If-Last-Modified` header, an `HTTP 304 Not Modified` can be returned without any payload.

Here's the diagram to illustrate the flow:


Client
Server
GET /foo
200 OK
Last-Modified: Mon, 24 Aug 2020 22:35:02 GMT
GET /foo
If-Modified-Since: Mon, 24 Aug 2020 22:35:02 GMT
304 Not Modified

As you can see it's quite similar to `Etag` and `If-None-Match`, but with timestamps instead of a content fingerprint.

> Varnish supports both `Etag`/`If-None-Match` and `Last-Modified`/`If-Modified-Since`. I personally prefer using *Etags* because it's more precise, but both mechanisms do the job just fine.

3.5.9 Conditional range requests

In one of the previous sections we talked about *range requests*. The goal is to receive partial content by requesting one or more *byte ranges* from a resource.

The client issues a `Range` header to indicate which portion of the content. When successful, an `HTTP 206 Partial Content` status is returned.

Range requests can also be done *conditionally* to ensure that the *up-to-date* version of a *byte range* is fetched.

The `If-Range` header can be used for validation purposes. This header either contains an `Etag` value or a `Last-Modified` value. If the `If-Range` matches the `Etag` or `Last-Modified` header, an `HTTP 206 Partial Content` is returned. If there's no match, the full payload is sent via an `HTTP 200 OK` status code.

Here's an example:

```
HTTP/1.1 200 OK
Etag: "5985cb907f843bb60f776d385eea6c82"
Accept-Ranges: bytes
Content-Length: 43
```

The response contains an `Etag` that can be used for conditional requests

- Because of the `Accept-Ranges: bytes` header, we know the server supports *range requests*
- The `Content-Length` header indicates the size response, which is *43 bytes*. This is the upper limit that can be used for *range requests*
- Because both the `Etag` and `Accept-Ranges: bytes` headers are there, we know we can perform *conditional range requests*

Here's such a *conditional range request*:

```
GET / HTTP/1.1
Range: bytes=0-19
If-Range: "5985cb907f843bb60f776d385eea6c82"
```

This *conditional range request* will retrieve the *first 20 bytes* from the `/` resource, but will only do this if the `Etag` matches `"5985cb907f843bb60f776d385eea6c82"`.

If that is the case, the following response can be expected:

```
HTTP/1.1 206 Partial Content
Etag: "5985cb907f843bb60f776d385eea6c82"
Accept-Ranges: bytes
Content-Range: bytes 0-19/43
Content-Length: 20
```

If the *Etag* doesn't match, this means the content has changed. As a consequence, no *partial content* can be returned, and instead the full payload is returned:

```
HTTP/1.1 200 OK
Etag: "9985cb2a7f8413b60f7789aa5eea6c41"
Accept-Ranges: bytes
Content-Length: 43
```

Because Varnish only has built-in *range support* on the *client side*, conditional range requests are only performed on the *client side*.

3.6 Compression

In an attempt to reduce the size of the response payload over the wire, various compression algorithms are used to compress the response body.

The most popular compression algorithm by far is *gzip*. HTTP has the mechanisms in place to perform content negotiation, and to request either the *plain-text* or the *compressed* version.

3.6.1 Content negotiation

In terms of *content negotiation*, the browser can advertise its *content encoding capabilities* by issuing an `Accept-Encoding` header containing the compression/encoding algorithms it supports.

Here's what my browser advertises:

```
Accept-Encoding: gzip, deflate, br
```

Basically, my browser supports `gzip` compression, `deflate` compression, and `br` compression. `br` is short for *Brotli*.

A server can choose one of the supported compression algorithms and return the algorithm it used for compression in the form of a `Content-Encoding` header.

Because *gzip* is the most popular web compression algorithm, web servers are likely to return the following response header:

```
Content-Encoding: gzip
```

3.6.2 Gzip compression in Varnish

Varnish natively supports *gzip compression* and encourages servers to send compressed responses to Varnish.

It does so by modifying the `Accept-Encoding` header to `Accept-Encoding: gzip` for every miss, regardless of the `Accept-Encoding` value that the client may have set.

Responses will be stored in cache in a compressed format. If it turns out that the client doesn't support *gzip*, Varnish will decompress the response *on-the-fly*.

Because *gunzip* is very fast, it is more efficient to decompress *on-the-fly* than to store two versions of the object in cache.

When a client doesn't support *gzip*, the *plain-text* version is returned, the `Content-Encoding: gzip` header is stripped off, and *Etags* get weakened.

When a *gzip'ed* object is stored in cache, a `Vary: Accept-Encoding` header will be returned to the client. Any attempt by the origin to issue a `Vary: Accept-Encoding` will be ignored because only the compressed version is kept.

> If the origin doesn't respond with a compressed response, Varnish will trust that it's because the compression wasn't worth it. This is the case for images, video and other binary resources. In such cases, the object is stored as-is, and user requests for compression will be ignored. The next section explains how to override this.

To explicitly disable *gzip support* in Varnish, the `http_gzip_support` runtime parameter can be disabled.

3.6.3 Gzip and VCL

Even though Varnish handles *gzip compression* behind the scenes, the *VCL* programming language still allows you to override the default behavior.

Imagine that your origin server doesn't support *gzip*, but you still want to serve compressed content. In that case you can use the following *VCL snippet*:

```
sub vcl_backend_response {
    if (beresp.http.content-type ~ "text") {
        set beresp.do_gzip = true;
    }
}
```

As you can see, there is an *if-condition* in there to ensure that only *plain-text* content gets compressed. Binary content, such as *JPEG images*, shouldn't be compressed.

You can also decompress *gzip'ed* content in *VCL* by setting `set beresp.do_gunzip = true;`. You can even check whether or not the client supports *gzip* through the `req.can_gzip` variable, which returns a *boolean*.

3.6.4 Brotli compression in Varnish

As of version *6.0.6r10* Brotli compression is supported in *Varnish Enterprise*. It is not available by default but requires `vmod_brotli` to be initialized.

The module can handle *brotli-compressed responses* from the origin, but it can also turn *gzip-compressed data* into Brotli.

Here's a *VCL example* that will store *gzip-compressed* or plain-text objects into *brotli-compressed* objects:

```
vcl 4.1;

import brotli;

sub vcl_init {
  brotli.init(BOTH, transcode = true);
}

sub vcl_backend_response {
  if (beresp.http.content-encoding ~ "gzip" ||
      beresp.http.content-type ~ "text") {
    brotli.compress();
  }
}
```

By initializing `vmod_brotli` with `BOTH` as the encoding value, it can normalize the `Accept-Encoding` request header and support plain-text encoding, gzip encoding and Brotli encoding.

By setting the `transcode` argument to `true`, the module will decompress objects when the client doesn't support Brotli, or transcode the object to gzip when the client only supports gzip.

3.7 Content streaming

Streaming in an HTTP context is often associated with *video streaming*. And although *OTT video streaming* is an important use case for Varnish, this section is not about that.

In *chapter 10* we will be talking about *OTT video streaming* in detail.

Here, it means that Varnish avoids buffering, in the sense that it doesn't need to receive the full response from the origin before starting to send it to the user.

Instead, Varnish can start *streaming* the origin data to users as soon as it's available. This can significantly reduce latency when delivering live video, where certain origins can start delivering video chunks before they are completely processed. If Varnish were to buffer the chunk, the low-latency benefit of the origin would be lost.

3.7.1 Chunked transfer encoding

An *HTTP server* that wants to send the response in chunks, which implies not sending a `Content-Length` header, signals this by sending the following header to the client:

```
Transfer-Encoding: chunked
```

When all the headers have been sent to the client, the server starts sending HTTP chunks.

Each chunk is prefixed by its chunk length followed by a `\r\n` sequence, then there's the chunk itself, also followed by a `\r\n` sequence. A web server sending a chunked response will typically write one chunk at a time to the network socket, and this flushes the chunk to the client. Then it's a matter of repeating the process until all chunks are sent. Finally the server sends a *zero-length chunk* to mark the end of the transaction, and the HTTP connection can be used to request more resources.

This may sound very confusing, so here's an example:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

8\r\n
Varnish\n\r\n
```

```
9\r\n
supports\n\r\n
8\r\n
chunked\n\r\n
9\r\n
transfer\n\r\n
9\r\n
encoding\n\r\n
0\r\n
\r\n
```

The *chunk length* consists of the number of characters in each chunk, but the `\r\n` shouldn't be accounted for. However, any new lines that are part of the chunk output should be part of the *chunk length*.

> In this case `Varnish` is seven characters long, but the new line results in a *chunk length* of eight.

The output of this HTTP response will be:

```
Varnish
supports
chunked
transfer
encoding
```

In total there are five chunks that are flushed to the client one at a time unless the kernel helpfully combines some of the chunks into a single *TCP package*. If we know that it takes one second for each chunk to be rendered, output will start appearing as of second one.

If this were done using regular *buffered output*, the client would have to wait for five seconds before the output were to appear.

For time-consuming processes, *content streaming* using *chunked transfer encoding* will have a positive impact on the quality of experience for the end-user.

3.7.2 Streaming support in Varnish

Streaming delivery support in *Varnish* goes beyond the support for *chunked transfer encoding*. When *Varnish* fetches from a server, it will start sending the response body to the clients while still fetching from the backend, independently of *chunked transfer encoding* being used or not.

When *Varnish* does not know the length of the body while streaming because the server is using *chunked transfer encoding*, and the fetch is still ongoing, it will use *chunked transfer encoding* when sending the response to clients.

Once the response is fully processed, *Varnish* will know the *content length*, and no longer use chunked encoding when sending the response to new clients.

This means that a *cache miss* will be streamed to the client using the `Transfer-Encoding: chunked` response header. But for the next request, if it is a hit, and the object has been fully fetched from the backend, the entire response body is sent at once, and a `Content-Length` header is included.

Varnish switches to `Content-Length` as soon as it can because it is *cheaper* in terms of resource usage. When the entire object is in memory, *Varnish* can send all of the data to the kernel in a single call, and the overhead associated with *chunked encoding* is eliminated.

In *VCL*, you can enable or disable streaming by toggling the value for the `beresp.do_stream` variable. The default value for this variable is `true`.

Here's an example of a situation in which streaming for streamable content is disabled:

```
sub vcl_backend_response {
    if(bereq.url == "/my-page") {
        set beresp.do_stream = false;
    }
}
```

This snippet will disable streaming if the *request URL* is `/my-page`.

You can also check *on delivery* whether or not streaming was used by reading the value of `resp.is_streaming`, which will return a *boolean*.

It happens that backends suddenly fail, or stop sending data in the middle of a transaction. If *Varnish* is streaming, and data stops coming from the backend, it can only signal this to the client by hanging up, leaving the client with a partial response. On the other hand, if streaming has been disabled for the given transaction, *Varnish* will be able to send an *HTTP 503* error code to the client when it realizes that the full response cannot be sent to the client.

3.8 Summary

HTTP is undeniable: obviously web browsers use HTTP to retrieve website data. HTTP is also a very dominant protocol for *server-to-server* communication, with *HTTP-based APIs* as the main driver.

Whereas in the past applications would use a custom protocol, HTTP is now the standard for *client-server communication*.

We used to say that *TCP/IP* powered the internet, but for the last decade, people are aiming higher in the network stack, and have made *HTTP* their protocol of choice.

As the internet changes, and as HTTP is used for a lot of new and challenging applications, HTTP is slowly adapting to these modern-day needs.

From a *web performance* point of view, HTTP already comes with quite a bit of syntax to control how responses can be cached.

Varnish, as all good caching citizens, will comply with these standards and best practices.

This chapter was all about HTTP, and how it can be leveraged for caching. Not just for the sake of it, but to illustrate how little customization is required in Varnish to gain control over your content delivery flow.

In the next chapter, we'll talk about the *Varnish Configuration Language*, and how it can be used to customize the behavior of Varnish. But the most important lesson of this chapter is: the less *VCL* you have to write, the better. Because a lot of it is covered by HTTP already.

From the `Cache-Control` header, to *cache variations*, and even streaming, compression, and conditional requests: HTTP already offers so many valuable caching features.

But in most *real-world scenarios*, HTTP doesn't have all the answers. And that's where *VCL* comes into play. Get ready for *chapter 4*.

Chapter 4: The Varnish Configuration Language

Welcome to *chapter 4*, in which we'll discuss the *Varnish Configuration Language*, or *VCL* as we'll refer to it.

In *chapter 1*, we explained what *Varnish* is. In *chapter 2*, we went into detail about what's new in *Varnish 6*. In *chapter 3*, we showed you how to control Varnish's behavior using *HTTP's built-in caching mechanisms*.

HTTP has solid caching features, as you're aware after having read *chapter 3*. But in *real-world scenarios* you'll often fall short, and you need some sort of configuration mechanism that allows you to customize the system beyond what's possible in standard HTTP.

Varnish can do even better: instead of a configuration file, you get a programming language. If your question is: "Does this make Varnish edge computing technology?", the answer is definitely "yes".

4.1 What is VCL again?

VCL is a *domain-specific* language, meaning that it can only be used for *Varnish* and has no other application.

The *Varnish Configuration Language* has a *curly braces* syntax style and feels familiar to languages like *C*, *C++*, *C#*, *Java*, and many more.

VCL code is written in a *VCL file*, which is picked up by the `varnishd` process on startup. The `-f` runtime parameter refers to the *VCL file* that needs to be loaded.

Upon startup, *VCL code* is translated into *C code*, which in turn gets compiled. The *shared object* that results from this compilation process is loaded into the *Varnish server process*.

The fact that the code gets compiled and is not interpreted at runtime makes *VCL* a very fast programming language. This is not a side effect, but a design goal, since *Varnish* is all about speed and scalability.

VCL is not a typical *top-down programming language*. *VCL* is a language that hooks into various *states* of a *finite state machine*. This allows *VCL* to extend the *built-in VCL* behavior of Varnish.

We'll cover the *finite state machine* in the coming sections, and the *built-in VCL code* will be covered at the end of this chapter.

VCL is primarily used for *request and response manipulation*, *backend routing*, and the execution of *caching policies*.

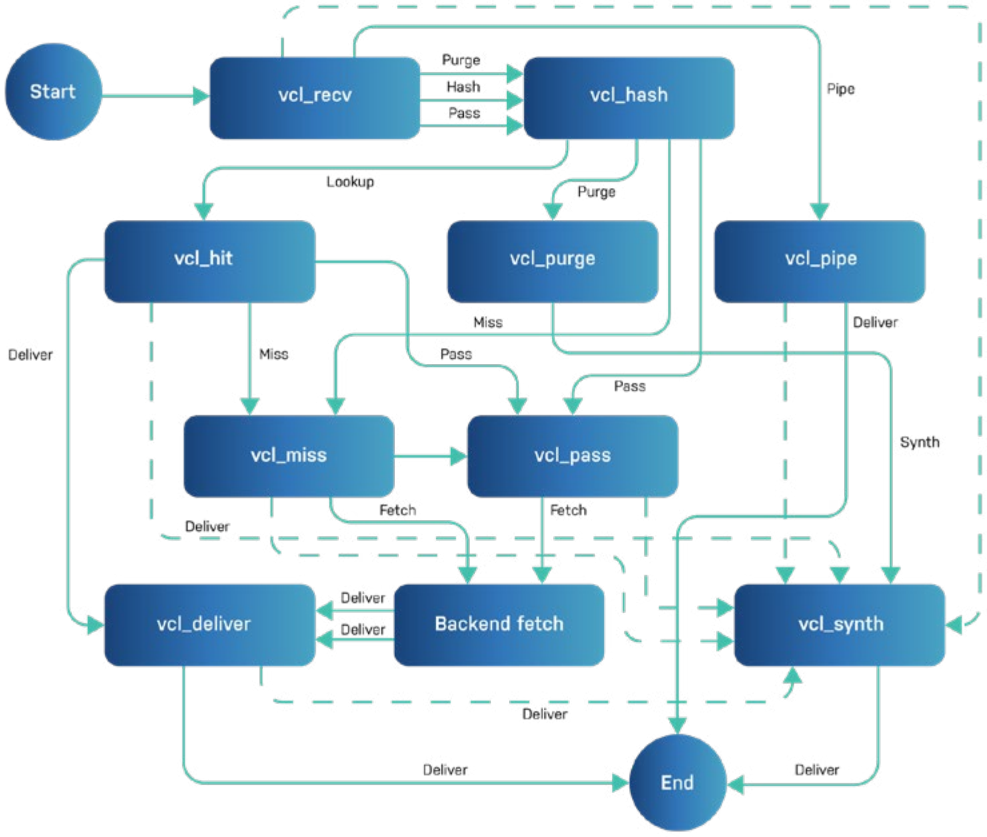
In more advanced use cases, *VCL* will be used to control *web application firewall* features, to offload authentication, to parse and process *edge side includes*, and to modify the response body based on interaction with *third-party* services. And that is what we call *edge computing*.

4.2 The finite state machine

As mentioned earlier, the *VCL* language allows you to extend the behavior of various Varnish states that are part of the *Varnish finite state machine*.

This probably makes sense to some extent, but without a visual representation, it is a tough concept to grasp.

Here's the flowchart that describes the states and the state transitions of this *finite state machine*. The diagram below represents the interaction between a client and *Varnish*.



Varnish's finite state machine on the client side

4.2.1 The client-side flow

It all starts when *Varnish* receives a request from a client. `vcl_recv` gets called, and depending on certain request criteria, a couple of different actions can be taken.

These actions control which path is taken through the *finite state machine*:

- The `vcl_recv`, `vcl_hash`, `vcl_hit`, `vcl_deliver` path represents the desired outcome: a *cache hit*.
- The `vcl_recv`, `vcl_hash`, `vcl_miss`, `Backend fetch`, `vcl_deliver` path represents an acceptable outcome: a *cache miss*.
- The `vcl_recv`, `vcl_hash`, `vcl_pass`, `Backend fetch`, `vcl_deliver` path represents an undesirable outcome: bypassing the cache.
- The `vcl_recv`, `vcl_pipe` path represents an escape plan: bypassing *HTTP* entirely and switching to *TCP*.
- The `vcl_recv`, `vcl_hash`, `vcl_purge`, `vcl_synth` path represents a *cache purge*, which explicitly removes an object from cache.
- The dotted lines represent potential transitions to return *synthetic output* at any point in time.

Remember: *cache misses* aren't a bad thing. A *miss* is just a *hit* that didn't happen yet.

There's always the incoming request that triggers the start of the flow, but there must also be something that ends the transaction. In *HTTP*, we always expect a response to be returned.

From VCL, we can `return(abandon)`, which will just drop the connection. This can be desirable in some cases but breaks the HTTP transaction, and that's another story.

And that's how the *finite state machine* ends the transaction: by delivering a response to the client. It could be a cached object, it could be a backend fetch, or it could just be synthetic output.

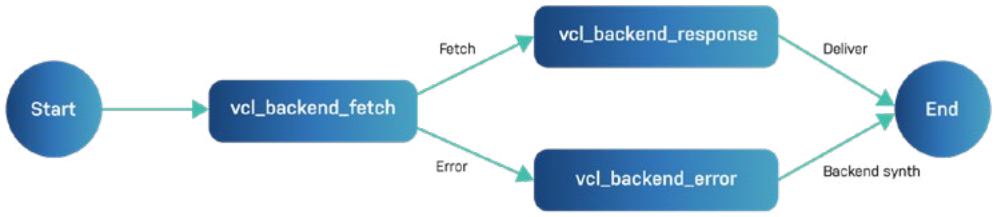
What you don't see in this diagram is the *backend flow*. When there is a cache miss, or the cache is bypassed, you'll need to connect to the origin and fetch the result. In this diagram, backend interaction was abstracted into a single *backend-fetch* state.

Let's have a look at the *backend flow* in some more detail.

4.2.2 The backend flow

The *backend flow* represents the communication between *Varnish* and the *origin server*.

As you can see in the diagram below, the flow is a lot simpler compared to the *client-side flow*:



Varnish's finite state machine on the backend side

In `vcl_backend_fetch`, the request to the backend is prepared, and the original *client request* is turned into a *backend request*.

Depending on what happens on the backend, you either end up in `vcl_backend_response` when the request is successfully processed, or in `vcl_backend_error` when an error occurs.

In `vcl_backend_response` a number of checks happen to decide whether or not to cache the response. Eventually the response is sent back to the *client-side logic* of Varnish, which will send the response to the client.

The `vcl_backend_error` stage is reached when *Varnish* fails to connect to a backend, when a backend is considered as *sick*, or when the backend doesn't respond in time. You can also reach this stage from `vcl_backend_fetch` or `vcl_backend_response` by using a `return(error)` statement.

The result is that a *synthetic error* is returned and sent back to Varnish's *client-side logic* with an `HTTP 503 Service Unavailable` error.

Surprisingly, other *500-range* errors that were received from the backend aren't considered errors. They *can* be cached, `vcl_backend_error` is not triggered, and the response is sent to the client without any interference from *Varnish*.

Whether you have a successful response, or an error, a *backend response* is returned, which in its turn will be sent to the client.

4.3 Hooks, subroutines, and built-in VCL

The previous section featured the *Varnish finite state machine*. Every state has a corresponding subroutine that allows you to *hook* into that state to modify its behavior.

In this section, we'll cover the various *subroutines* and their corresponding *VCL code*, and we'll explain how this code fits into the *Varnish finite state machine*.

> The *VCL code* you're about to see is all part of what we call the *built-in VCL*. We've covered this behavior in the previous chapter; now you'll see the actual code.
>
> Remember: even if this code is not part of your *VCL file*, it will still be executed by *Varnish* if you don't perform an explicit *return call*.

4.3.1 vcl_recv

`vcl_recv` is the first subroutine that is used in the *built-in VCL*. It hooks into the *request-handling logic*. Based on certain criteria, it transitions to another state by returning a specific action.

Let's have a look at the `vcl_recv` *VCL code*:

```
sub vcl_recv {
    if (req.method == "PRI") {
        /* This will never happen in properly formed traffic (see:
RFC7540) */
        return (synth(405));
    }
    if (!req.http.host &&
      req.esi_level == 0 &&
      req.proto ~ "^(?i)HTTP/1.1") {
        /* In HTTP/1.1, Host is required. */
        return (synth(400));
    }
    if (req.method != "GET" &&
      req.method != "HEAD" &&
      req.method != "PUT" &&
      req.method != "POST" &&
      req.method != "TRACE" &&
      req.method != "OPTIONS" &&
      req.method != "DELETE" &&
      req.method != "PATCH") {
        /* Non-RFC2616 or CONNECT which is weird. */
        return (pipe);
    }

    if (req.method != "GET" && req.method != "HEAD") {
        /* We only deal with GET and HEAD by default */
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie) {
        /* Not cacheable by default */
        return (pass);
    }
    return (hash);
}
```

Error cases

There are two *error cases* that will result in *synthetic responses* being returned:

When the *request method* is `PRI`, this means an *HTTP/2* request is received, whereas *Varnish* wasn't configured to handle *HTTP/2*. This is not supposed to happen, and a `HTTP 405 Method Not Allowed` error is *synthetically* returned.

Here's the *VCL code* for that:

```
if (req.method == "PRI") {
    /* This will never happen in properly formed traffic (see:
RFC7540) */
    return (synth(405));
}
```

The other *error case* is when a top-level `HTTP/1.1` request is made without a `Host` header. This goes against the rules of the protocol and results in an `HTTP 400 Bad Request` error being returned *synthetically*.

Here's the corresponding *VCL code*:

```
if (!req.http.host &&
  req.esi_level == 0 &&
  req.proto ~ "^(?i)HTTP/1.1") {
    /* In HTTP/1.1, Host is required. */
    return (synth(400));
}
```

We referred to the term *top-level request*. This is the main *HTTP request*. Varnish can also trigger *subrequests*, which are part of the *ESI parsing logic*.

The *top-level check* is done by checking the value of the `req.esi_level`

To pipe or not to pipe

The next check that is performed in `vcl_recv` is also related to the *request method*. There is a series of *HTTP request methods* that Varnish accepts. If the header that is received doesn't match this list, then `return(pipe)` is executed, as illustrated below:

```
if (req.method != "GET" &&
  req.method != "HEAD" &&
  req.method != "PUT" &&
  req.method != "POST" &&
  req.method != "TRACE" &&
  req.method != "OPTIONS" &&
  req.method != "DELETE" &&
  req.method != "PATCH") {
    /* Non-RFC2616 or CONNECT which is weird. */
    return (pipe);
}
```

Piping means that *Varnish* no longer considers this an *HTTP request*. Instead, it just treats the data as *TCP* and shuffles the payload over the wire, without further interference. If dealing with *HTTP requests*, always consider using a *pass* instead of a *pipe*, as piping relinquishes your ability to manipulate the transaction in further steps, and your logs will be blind to the backend response.

Only GET and HEAD

Varnish follows *HTTP best practices*. When it comes to caching, only *idempotent requests* may be cached. This means: request methods that don't explicitly change the state of the resource.

As a result, `GET` and `HEAD` are the only two cacheable *request methods*. This rule is enforced using the following *VCL snippet* in `vcl_recv`:

```
if (req.method != "GET" && req.method != "HEAD") {
    /* We only deal with GET and HEAD by default */
    return (pass);
}
```

So if the *request method* is for example `POST`, the `return(pass)` logic will kick in, and you'll be sent to the `vcl_pass` subroutine. Requests that end up in `vcl_pass` will bypass the cache, and will result in a *backend fetch*.

Stateless

Stateful content is always difficult to cache. As mentioned in *chapter 3*, *cache variations* allow you to have multiple variations on the same resource. But when content is *for your eyes only*, usually you will not cache this content.

In *HTTP*, there are two common ways to keep track of state:

- Through a `Cookie` header, which contains *key-value pairs of user data*
- Through an `Authorization` header, which contains an *authentication token* that authorizes the client

Technically, the `Authorization` header isn't automatically conveying a state, but like a `Cookie`, it denotes a customization of the content and that without deeper knowledge, it may be dangerous to cache the data.

In `vcl_recv`, any request containing a `Cookie` header, or an `Authorization` will result in a `return(pass)` too. Here's the *VCL code* to prove it:

```
if (req.http.Authorization || req.http.Cookie) {
    /* Not cacheable by default */
    return (pass);
}
```

Anything else gets cached

So in the end, if you jumped through all those hoops, *Varnish* will consider your request *cacheable* and will look the corresponding object up in cache.

In *VCL*, this means performing a `return(hash)`, which is exactly what happens at the end of the `vcl_recv` subroutine.

Once a *hash* is created to identify the object in cache, it means that you have a stateless and idempotent request that complies with the *HTTP spec* in terms of the *request method* and the *host header*.

4.3.2 vcl_hash

Although the diagram of the *Varnish finite state machine* uses `vcl_hash` as a *point-of-entry* for many other states, there is only one *return statement* that is actually used in the *built-in VCL*, and that is `return(lookup)`.

Here's the *VCL*:

```
sub vcl_hash {
    hash_data(req.url);
    if (req.http.host) {
        hash_data(req.http.host);
    } else {
        hash_data(server.ip);
    }
    return (lookup);
}
```

This subroutine will use the `hash_data()` function to create the *hash* of the object that is requested.

As you know from *chapter 3*, the hash is composed using the *request URL* and the *host header*. If there is no *host header*, the *server IP address* will be used instead.

What happens next all depends on the result of `return(lookup)`:

- If the object is found, we'll end up in `vcl_hit`.
- If the object is found, but was marked as *uncacheable*, we will transition to `vcl_pass`.
- If the object is found, but the request was a *purge request*, we'll end up in `vcl_purge`.
- If the object is not found, we'll end up in `vcl_miss`.

4.3.3 vcl_hit

Whenever a requested object is found in cache, a transition will happen from `vcl_hash` to `vcl_hit`.

In the diagram, a multitude of *return actions* are available for this state. However, the *built-in VCL* only has two default outcomes for `vcl_hit`:

```
sub vcl_hit {
    if (obj.ttl >= 0s) {
        // A pure unadulterated hit, deliver it
        return (deliver);
    }
    if (obj.ttl + obj.grace > 0s) {
        // Object is in grace, deliver it
        // Automatically triggers a background fetch
        return (deliver);
    }
    // fetch & deliver once we get the result
    return (miss);
}
```

If it turns out the object still has some *TTL* left, the object will be delivered. This means we'll transition to `vcl_deliver`.

If the *TTL* has expired, but there's still some *grace* left, the object will also be delivered while a background fetch happens for revalidation. This is the typical *stale while revalidate* behavior we discussed in the previous chapter.

If none of these conditions apply, we can conclude that the object has expired without any possibility of delivering a stale version. This is the same thing as a *cache miss*, so we fetch and deliver the new version of the object.

A dirty little secret about vcl_hit

A dirty little secret about the *VCL code* in `vcl_hit` is that it doesn't really behave the way it is set up. The `if (obj.ttl + obj.grace > 0s) {}` conditional will always evaluate to `true`.

In reality, the *built-in VCL* for `vcl_hit` could be replaced by the following snippet:

```
sub vcl_hit {
    return (deliver);
}
```

The *VCL* is just there to show the difference between a *pure hit* and a *grace hit*.

In newer *Varnish* versions, it's actually what `vcl_hit` looks like, as *grace* is handled internally.

4.3.4 vcl_miss

There's not a lot to say about `vcl_miss`, really. Although a transition to `vcl_pass` is supported, the *built-in VCL* just does a `return(fetch)` for `vcl_miss`.

Here's the *VCL*:

```
sub vcl_miss {
    return (fetch);
}
```

4.3.5 vcl_purge

When you enter the `vcl_purge` stage, it means that you called a request to purge an object from cache. This is done by calling `return(purge)` in `vcl_recv`.

Based on the URL and hostname of the corresponding request, the object hash is looked up in cache. If found, all objects under that hash are removed from cache, and the transition to `vcl_purge` happens. If the hash didn't exist we still transition to `vcl_purge` because the outcome is the same: not having an object in cache for that hash.

As illustrated in the *VCL example* below, `vcl_purge` will return a *synthetic response*:

```
sub vcl_purge {
    return (synth(200, "Purged"));
}
```

The response itself is very straightforward: `HTTP/1.1 200 Purged`.

4.3.6 vcl_pass

A lot of people assume that there is just *hit or miss* when it comes to caches. *Hit or miss* is the answer to the following question:

Did we find the requested object in cache?

But there are more questions to ask. The main question to ask beforehand is:

> Do we want to serve this object from cache?

And that is where *pass* enters the proverbial *chat*.

As you know from the *built-in VCL*: when you don't want something to be served from cache, you just execute `return(pass)`. This is where you enter `vcl_pass`.

Apart from its intention, the *built-in VCL* implementation of `vcl_pass` is identical to `vcl_miss`: you perform a `return(fetch)` to fetch the content from the origin.

Here's the *VCL*:

```
sub vcl_pass {
    return (fetch);
}
```

And during the *lookup stage*, when a *hit-pass* object is found, instead of a regular one, an immediate transition to `vcl_pass` happens as well.

4.3.7 vcl_pipe

The *built-in VCL* code for `vcl_pipe` has a big disclaimer in the form of a comment:

```
sub vcl_pipe {
    # By default Connection: close is set on all piped requests, to
stop
    # connection reuse from sending future requests directly to the
    # (potentially) wrong backend. If you do want this to happen, you
can undo
    # it here.
    # unset bereq.http.connection;
    return (pipe);
}
```

The implementation, and the comment are a bit special. But then again, *piping* only happens under special circumstances.

The fact that you ended up in `vcl_pipe`, means that Varnish is under the impression that the request is not an HTTP request. We've learned from the *built-in VCL* that `return(pipe)` is used when the *request method* is not recognized.

Piping steps away from the *layer 7 HTTP implementation of Varnish* and goes all the way down to *layer 4*: it treats the incoming request as *plain TCP*, it no longer processes HTTP, and just shoves the *TCP packets* over the wire.

When the transaction is complete Varnish will close the connection with the origin to prevent other requests from reusing this connection.

A lot of people think that `return(pass)` and `return(pipe)` are the same in terms of behavior and outcome. That's clearly not the case, as `vcl_pass` is still aware of the *HTTP context*, whereas `vcl_pipe` has no notion of HTTP.

4.3.8 vcl_synth

You enter the `vcl_synth` state when you execute a `return(synth())` using the necessary function parameters for `synth()`.

As mentioned before, *synthetic responses* are HTTP responses that don't originate from a backend response. The output is completely fabricated within *Varnish*.

In the *built-in VCL*, the `vcl_synth` subroutine adds some markup to the output:

```
sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";
    set resp.body = {"<!DOCTYPE html>
<html>
  <head>
    <title>"} + resp.status + " " + resp.reason + {"</title>
  </head>
  <body>
    <h1>Error "} + resp.status + " " + resp.reason + {"</h1>
    <p>"} + resp.reason + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + req.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"};
    return (deliver);
}
```

The assumption of the *built-in VCL* is that the output should be in *HTML*, which is also reflected in the `Content-Type` response header that is set.

Imagine the following *synth call*:

```
return(synth(200,"OK"));
```

The corresponding *synthetic response* would be the following:

```
HTTP/1.1 200 OK
Date: Tue, 08 Sep 2020 07:34:25 GMT
Server: Varnish
X-Varnish: 5
Content-Type: text/html; charset=utf-8
Retry-After: 5
Content-Length: 224
Accept-Ranges: bytes
Connection: keep-alive

<!DOCTYPE html>
<html>
  <head>
    <title>200 OK</title>
  </head>
  <body>
    <h1>Error 200 OK</h1>
    <p>OK</p>
    <h3>Guru Meditation:</h3>
    <p>XID: 5</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
```

The `Content-Type` and the `Retry-After` headers were set in `vcl_synth`, whereas all other headers are set behind the scenes by *Varnish*.

When using the *built-in VCL* untouched, this is the *HTML output* that will be returned to the client.

4.3.9 vcl_deliver

Before a response is served back to the client, served from cache or from the origin, it needs to pass through `vcl_deliver`.

The *built-in VCL* is not exciting at all:

```
sub vcl_deliver {
    return (deliver);
}
```

Most people use `vcl_deliver` to decorate or clean some *response headers* before delivering the content to the client.

4.3.10 vcl_backend_fetch

When an object cannot be served from cache, a *backend fetch* will be made. As a result, you'll end up in `vcl_backend_fetch`, where the original request is converted into a *backend request*.

Here's the *built-in VCL*:

```
sub vcl_backend_fetch {
    if (bereq.method == "GET") {
        unset bereq.body;
    }
    return (fetch);
}
```

The fact that `return(fetch)` is called in this subroutine is not surprising at all. But what is surprising is that the *request body* is removed when a `GET` request is made.

Although a *request body* for a `GET` request is perfectly allowed in the *HTTP spec*, *Varnish* decides to strip it off.

The reason for that makes a lot of sense from a caching point of view: if there's a *request body*, the *URL* is no longer the only way to uniquely identify the object in cache. If the *request body* differs, so does the object. To make this work, one would have to perform a *cache variation* on the request body, which could seriously decrease the hit rate.

Since *request bodies* for `GET` requests aren't all that common, *Varnish* protects itself by conditionally running `unset bereq.body`.

Also, if the request is a *cache miss*, *Varnish* will automatically turn the request into a `GET` request. If the request was a `HEAD` request, this is what we expect because *Varnish* must have the response body to operate correctly. However, if the request was a `POST` request or something else, and you want to cache the response, you must save the request method in a header and put it back in this subroutine.

You probably noticed that we're using the `bereq` object to identify the request, instead of the `req` object we used earlier. That's because we're now in *backend context*, and the original request has been copied over into the *backend request*. You'll learn all about objects and variable in *VCL* later in this chapter.

4.3.11 vcl_backend_response

The `vcl_backend_response` subroutine is quite a significant one: it represents the state after the origin successfully returned an *HTTP reponse*. This means the request didn't result in a *cache hit*.

It is also the place where *Varnish* decides whether or not to store the response in cache. Based on the *built-in VCL code* below, you'll see that there's some decision-making in place:

```
sub vcl_backend_response {
    if (bereq.uncacheable) {
        return (deliver);
    } else if (beresp.ttl <= 0s ||
      beresp.http.Set-Cookie ||
      beresp.http.Surrogate-control ~ "(?i)no-store" ||
      (!beresp.http.Surrogate-Control &&
        beresp.http.Cache-Control ~ "(?i:no-cache|no-store|private)")
||
      beresp.http.Vary == "*") {
        # Mark as "Hit-For-Miss" for the next 2 minutes
        set beresp.ttl = 120s;
        set beresp.uncacheable = true;
    }
    return (deliver);
}
```

Uncacheable

There are two ways to mark an object as uncacheable. The first and more common way is to `set beresp.uncacheable = true;`. This marks the object as *hit-for-miss*.

You can also use the `return(pass)` syntax in this subroutine, which marks the object as *hit-for-pass*. *Hit-for-pass* and *hit-for-miss* are very similar in that they both instruct Varnish that the current object is not to be inserted into cache and to disable request serialization for future requests. The difference is that *hit-for-miss* is allowed to change its mind and insert a cacheable object into cache in the future. *Hit-for-pass* cannot: this object can never be cached, now, or in the future. This trade-off gives *hit-for-pass* slightly better performance when dealing with uncacheable objects.

The *built-in VCL* will perform a series of checks to decide whether or not the response is cacheable.

If it turns out it is not, the `set beresp.uncacheable = true;` logic is triggered, which marks the object as *hit-for-miss*.

As explained earlier in the book, we're caching the decision not to cache, which prevents future requests for this object ending up on the waiting list.

And `vcl_backend_response` checks for uncacheable objects with the following *built-in VCL code*:

```
if (bereq.uncacheable) {
    return (deliver);
}
```

This logic can be triggered when a `return(pass)` is called in the *client-side logic*, or for a *hit-for-pass* object. But by default, we don't perform *hit-for-pass*, but *hit-for-miss*, which is a more forgiving approach.

Zero TTL

The *built-in VCL* will make a response uncacheable when the *TTL* is zero (or less).

This can be caused by three things:

- `set beresp.ttl = 0s` ended up in the *VCL file*, without performing a `return(deliver)`.
- The `max-age` or `s-maxage` value of the `Cache-Control` header was set to zero.
- The `Expires` header contains a timestamp in the past.

This is the check that happens in the *if-statement* to validate the *TTL*:

```
if(beresp.ttl <= 0s) {
```

As expected, this is the result:

```
set beresp.uncacheable = true;
```

A cookie was set

When the origin adds a `Set-Cookie` header to the response, it implies that the state of a cookie needs to change.

Whenever state is present, let alone changed, *Varnish* decides to bypass the cache. Both at the *client side*, and the *backend side*.

This is the cookie check that happens in the *if-statement* of the *built-in VCL*:

```
if(beresp.http.Set-Cookie) {
```

And again, this is the outcome:

```
set beresp.uncacheable = true;
```

Surrogate control

A `Surrogate-Control` header takes precedence over any other caching header. When such a header is set and its value contains `no-store`, the *built-in VCL* will make the response *uncacheable*.

Here's the *if-check*:

```
if(beresp.http.Surrogate-control ~ "(?i)no-store") {
```

And once again, here's the outcome:

```
set beresp.uncacheable = true;
```

Cache control says no

When your response doesn't contain a `Surrogate-Control` header, the *built-in VCL* will check if your response has a `Cache-Control` header.

If that is the case, the *built-in VCL* will make the response uncacheable if the `Cache-Control` header contains one of the following statements:

- `no-cache`
- `no-store`
- `private`

If you went through *chapter 3*, you already know about this. So, here's the *VCL code* to perform the check:

```
if(!beresp.http.Surrogate-Control &&
    beresp.http.Cache-Control ~ "(?i:no-cache|no-store|private)") {
```

The outcome is:

```
set beresp.uncacheable = true;
```

Vary all the things

Cache variations are good, but as with all things in life, you shouldn't exaggerate.

If you *vary* on all headers, there's no point caching the response, which is exactly what the *built-in VCL* thinks as well. Here's the code:

```
if(beresp.http.Vary == "*") {
```

Very predictably, the outcome is:

```
set beresp.uncacheable = true;
```

4.3.12 vcl_backend_error

When you reach `vcl_backend_error`, it means you didn't receive a valid HTTP response from the selected backend. There's a multitude of reasons why that could be the case. Not being able to connect to the backend is also part of that.

When this happens, we cannot return a regular response, and we have to return a *synthetic response* again. That's why the *built-in VCL* code for `vcl_backend_error` is nearly identical to the `vcl_synth` one.

The main difference is that `resp.http.Content-Type` becomes `beresp.http.Content-Type` because we're operating in a *backend-side context*, not a *client-side context*.

Here's the *built-in VCL code*:

```
sub vcl_backend_error {
    set beresp.http.Content-Type = "text/html; charset=utf-8";
    set beresp.http.Retry-After = "5";
    set beresp.body = {"<!DOCTYPE html>
<html>
  <head>
    <title>"} + beresp.status + " " + beresp.reason + {"</title>
  </head>
  <body>
    <h1>Error "} + beresp.status + " " + beresp.reason + {"</h1>
    <p>"} + beresp.reason + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + bereq.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"};
    return (deliver);
}
```

And here's the output you'll probably get when you run into a *backend error*:

```
HTTP/1.1 503 Backend fetch failed
Date: Tue, 08 Sep 2020 12:16:31 GMT
Server: Varnish
Content-Type: text/html; charset=utf-8
Retry-After: 5
X-Varnish: 5
Age: 0
Via: 1.1 varnish (Varnish/6.0)
Content-Length: 278
Connection: keep-alive

<!DOCTYPE html>
<html>
  <head>
    <title>503 Backend fetch failed</title>
  </head>
  <body>
    <h1>Error 503 Backend fetch failed</h1>
    <p>Backend fetch failed</p>
    <h3>Guru Meditation:</h3>
    <p>XID: 6</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
```

There are slightly more response headers, but apart from that, it's the same output template.

4.3.13 vcl_init

`vcl_init` is a subroutine that is called when the *VCL* is initialized, before requests are processed. It is the place where *VMODs* can be initialized, or where *VCL objects* can be created.

Out-of-the-box, no *VMODS* are initialized, and no objects are created. As a result, it just performs a `return(ok)`, as you can see in the example below:

```
sub vcl_init {
    return (ok);
}
```

4.3.14 vcl_fini

Whereas `vcl_init` is used when *VCL* is loaded, `vcl_fini` is used before the *VCL* is discarded.

This is the place where *VMODS* and *VCL objects* are cleaned up. By default we just perform a `return(ok)`. This is reflected in the example below:

```
sub vcl_fini {
    return (ok);
}
```

4.4 VCL syntax

All this talk about the illustrious *Varnish Configuration Language*, and yet it took us until this section to talk about the syntax. We'd apologize for this, but really it's all part of the plan.

Although this is a *domain-specific language* with no real other applications, the syntax is pretty easy to understand, particularly in the previous section where we showed the *built-in VCL code*, where it should make enough sense to comprehend.

In this section, we'll take a look at some of the basics of *VCL*. We won't focus too much on the *subroutines*, and the *finite state machine* because we've just done that. Let's just talk about how you can get things done within one of those *subroutines*.

4.4.1 VCL version declaration

Every *VCL file* starts with a version declaration. As of *Varnish 6*, the version declaration you'll want to use is the following:

```
vcl 4.1;
```

The *VCL version declaration* does not reflect the Varnish version it runs on, but instead ensures compatibility of the *VCL syntax*. In *Varnish 6*, *Unix domain sSocket* support introduced a backwards compatibility break: the `.path` variable in the backend declaration is not supported on older versions of Varnish and *VCL syntax version 4.0*.

The `vcl 4.0;` declaration will still work on *Varnish 6*, but it prevents some specific *Varnish 6* features from being supported.

4.4.2 Assigning values

The *VCL* language doesn't require you to program the full behavior, but rather allows you to extend pre-existing *built-in* behavior. Given this scope and purpose, the main objective is to set values based on certain conditions.

Let's take the following line of code for example:

```
set resp.http.Content-Type = "text/html; charset=utf-8";
```

It comes from the `vcl_synth` *built-in VCL* and assigns the content type `text/html; charset=utf-8` to the response HTTP header `resp.http.Content-Type`.

We're basically assigning a string to a variable. The assigning is done by using the `set` keyword. If we want to *unset* a variable, we just use the `unset` keyword.

Let's illustrate the `unset` behavior with another example from the *built-in VCL*:

```
unset bereq.body;
```

We're unsetting the `bereq.body` variable. This is part of the `vcl_backend_fetch` logic of the *built-in VCL*.

4.4.3 Strings

VCL supports various data types, but the *string type* is by far the most common.

Here's a conceptual example:

```
set variable = "value";
```

This is the easiest way to assign a string value. But as soon as you want to use newlines or double quotes, you're in trouble.

Luckily there's an alternative, which is called *long strings*. A *long string* begins with `{"` and ends with `"}`.

They may include newlines, double quotes, and other control characters, except for the *NULL* (`0x00`) character.

A very familiar usage of this is the *built-in VCL* implementation of `vcl_synth`, where the *HTML template* is composed using *long strings*:

```
set resp.body = {"<!DOCTYPE html>
<html>
  <head>
    <title>"} + resp.status + " " + resp.reason + {"</title>
  </head>
  <body>
    <h1>Error "} + resp.status + " " + resp.reason + {"</h1>
    <p>"} + resp.reason + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + req.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"};
```

There is also an alternate form of a long string, which can be delimited by triple double quotes, `"""..."""`.

This example also shows how to perform *string concatenation and variable interpolation*. Let's reimagine the `vcl_synth` example, and create a version using *simple strings*:

```
set beresp.body = "Status: " + resp.status + ", reason: " +
resp.reason";
```

And again we're using the `+`-sign for *string concatenation and variable interpolation*.

4.4.4 Conditionals

Although the *VCL* language is limited in terms of *control structures*, it does provide *conditionals*, meaning *if/else statements*.

Let's take some *built-in VCL* code as an example since we're so familiar with it:

```
if (req.method != "GET" && req.method != "HEAD") {
    /* We only deal with GET and HEAD by default */
    return (pass);
}
```

This is just a regular *if-statement*. We can also add an *else clause*:

```
if (req.method != "GET" && req.method != "HEAD") {
    /* We only deal with GET and HEAD by default */
    return (pass);
} else {
    return (hash);
}
```

And as you'd expect, there's also an *elseif clause*:

```
if (req.method == "GET") {
    return (hash);
} elseif (req.method == "HEAD") {
    return (hash);
} else {
    return (pass);
}
```

`elsif`,`elif` and `else if` can also be used as an equivalent for `elseif`.

4.4.5 Operators

VCL has a number of operators that either evaluate to `true` or to `false`:

- The `=` operator is used to assign values.
- The `==`, `!=`, `<`, `<=`, `>`, and `>=` operators are used to compare values.
- The `~` operator is used to match values to a *regular expression* or an *ACL*.
- The `!` operator is used for *negation*.
- `&&` is the *logical and operator*.
- `||` is the *logical or operator*.

And again the *built-in VCL* comes to the rescue to clarify how some of these operators can be used:

```
if (req.method != "GET" && req.method != "HEAD") {
    /* We only deal with GET and HEAD by default */
    return (pass);
}
```

You can clearly see the negation and the *logical and*, meaning that the expression only evaluates to `true` when condition one and condition two are false.

We've already used the `=` operator to assign values, but here's another example for reference:

```
set req.http.X-Forwarded-Proto = "https";
```

This example assigns the `https` value to the `X-Forwarded-Proto` request header.

A *logical or* looks like this:

```
if(req.method == "POST" || req.method == "PUT") {
    return(pass);
}
```

At least one of the two conditions has to be true for the expression to be `true`.

And let's end this part with a *less than or equals* example:

```
if(beresp.ttl <= 0s {
    set beresp.uncacheable = true;
}
```

4.4.6 Comments

Documenting your code with comments is usually a good idea, and *VCL* supports three different comment styles.

We've listed all three of them in the example below:

```
sub vcl_recv {
    // Single line of out-commented VCL.
    # Another way of commenting out a single line.
    /*
        Multi-line block of commented-out VCL.
    */
}
```

So you can use `//` or `#` to create a single-line comment. And `/* ... */` can be used for multi-line comments.

Pretty straightforward, not all that exciting, but definitely noteworthy.

4.4.7 Numbers

VCL supports numeric values, both *integers* and *real numbers*.

Certain fields are numeric, so it makes sense to assign *literal integer or real values* to them.

Here's an example:

```
set resp.status = 200;
```

But most variables are strings, so these numbers get cast into strings. For *real numbers*, their value is rounded to *three decimal places* (e.g. `3.142`).

4.4.8 Booleans

Booleans can be either `true` or `false`. Here's an example of a *VCL variable* that expects a boolean:

```
set beresp.uncacheable = true;
```

This example probably looks familiar. It comes from the *built-in VCL* and makes a response uncacheable.

When evaluating values of non-boolean types, the result can also be a boolean.

For example *strings* will evaluate to `true` or `false` if their existence is checked. This could result in the following example:

```
if(!req.http.Cookie) {
    //Do something
}

if(req.http.Authorization) {
    //Do something
}
```

Be aware that the header variable must be undefined or `unset` for it to be evaluated as `false`. If the header variable is defined with an empty value, it will evaluate as `true`.

Integers will evaluate to `false` if their value is `0`; the same applies to *duration types* when their values are *zero or less*.

Boolean types can also be set based on the result of another boolean expression:

```
set beresp.uncachable = (beresp.http.do-no-cache == "true");
```

4.4.9 Time & durations

Time is an absolute value, whereas a *duration* is a relative value. However, in *Varnish* they are often combined.

Time

You can add a duration to a time, which results in a new time. It admittedly sounds confusing, but here's some code to clarify this statement:

```
set req.http.tomorrow = now + 1d;
```

The `now` variable is how you can retrieve the current time in *VCL*. The `now + 1d` statement means we're adding a day to the current time. The returned value is also a *time type*.

But since we're assigning a *time type* to a *string field*, the time value is cast to a string, which results in the following string value:

```
Thu, 10 Sep 2020 12:34:54 GMT
```

Duration

As mentioned, durations are relative. They express a time change and are expressed numerically, but with a time unit attached.

Here are a couple of examples that illustrate the various time units:

- `1ms` equals *1 millisecond*.
- `5s` equals *5 seconds*.
- `10m` equals *10 minutes*.
- `3h` equals *3 hours*.
- `9d` equals *9 days*.
- `4w` equals *4 weeks*.
- `1y` equals *1 year*.

In *string context* their numeric value is kept, and the time unit is stripped off. This is exactly what a *real number* looks like when cast to a string. And just like *real numbers*, they are rounded to *three decimal places* (e.g. `3.142`).

Here's an example of a *VCL variable* that supports *durations*:

```
set beresp.ttl = 1h;
```

So this example sets the *TTL* of an object to *one hour*.

4.4.10 Regular expressions

Pattern matching is a very common practice in *VCL*. That's why *VCL* supports *Perl Compatible Regular Expressions (PCRE)*, and we can match values to a *PCRE regex* through the `~` operator.

Let's immediately throw in an example:

```
if(req.url ~ "^/[a-z]{2}/cart") {
    return(pass);
}
```

This example is matching the *request URL* to a *regex pattern* that looks for the shopping cart URL of a website. This URL is prefixed by *two letters*, which represent the user's selected language. When the *URL* is matched, the request bypasses the cache.

4.4.11 Backends

Varnish is a *proxy server* and depends on an *origin server* to provide (most of) the content. A *backend* definition is indispensable, even if you end up serving *synthetic content*.

The basics

This is what a *backend* looks like:

```
backend default {
    .host = "127.0.0.1";
    .port = "8080";
}
```

It has a *name*, `default` in this case, and uses the `.host` and `.port` properties to define how *Varnish* can connect to the origin server.

The first backend that is defined will be used by *Varnish*.

If you're not planning to use a backend, or if you are using a dynamic backend like `goto`, you'll have to define the following backend configuration:

```
backend default none;
```

This bypasses the requirement that you must define a single backend in your VCL.

Optional values

Backends also support the following options:

- `.connect_timeout` is how long to wait for a connection to be made to the backend.
- `.first_byte_timeout` is how long to wait for the first byte of the response.
- `.between_bytes_timeout` is the maximum time to wait between bytes when reading the response.
- `.last_byte_timeout` is the total time to wait for the complete backend response.
- `.max_connections` is the maximum number of concurrent connections Varnish will hold to the backend. When this limit is reached, requests will fail into `vcl_backend_error`.

Probes

Knowing whether or not a backend is healthy is important. It helps to avoid unnecessary outages and allows you to use a fallback system.

When using *probes*, you can perform *health checks* at regular intervals. The probe sets the internal value of the health of that backend to *healthy* or *sick*.

Backends that are *sick* always result in an *HTTP 503* error when called.

If you use `vmod_directors` to load balance with multiple backends, sick backends will be removed from the rotation until their health checks are successful and their state changes to *healthy*.

A *sick* backend will become *healthy* when a threshold of successful polls is reached within a polling window.

This is how you define a *probe*:

```
probe healthcheck {
}
```

Default values

The *probe data structure* has a bunch of attributes; even without mentioning these attributes, they will have a default behavior:

- `.url` is the *URL* that will be polled. The default value is `/`.
- `.expected_response` is the *HTTP status code* to that the probe expects. The default value is `200`.
- `.timeout` is the amount of time the probe is willing to wait for a response before timing out. The default value is `2s`.
- `.interval` is the *polling interval*. The default value is `5s`.
- `.window` is the number of polls that are examined to determine the backend health. The default value is `8`.
- `.initial` is the number of polls in `.window` that have to be successful before Varnish starts. The default value is `2`.
- `.threshold` is the number of polls in `.window` that have to be successful to consider the backend *healthy*. The default value is `3`.
- `.tcponly` is the mode of the probe. When enabled with `1`, the probe will only check for available TCP connections. The default value is `0`. This property is only available in *Varnish Enterprise*.

Extending values

You can start extending the probe by assigning values to these defaults.

Here's an example:

```
probe healthcheck {
    .url = "/health";
    .interval = 10s;
    .timeout = 5s;
}
```

This example will call the `/health` endpoint for polling and will send a health check every *ten seconds*. The probe will wait for *five seconds* before it times out.

Customizing the entire HTTP request

When the various probe options do not give you enough flexibility, you can even choose to fully customize the *HTTP request* that the probe will send out.

The `.request` property allows you to do this. However, this property is mutually exclusive with the `.url` property.

Here's an example:

```
probe healtcheck {
    .request =
        "HEAD /health HTTP/1.1"
        "Host: localhost"
        "Connection: close"
        "User-Agent: Varnish Health Probe";
    .interval = 10s;
    .timeout = 5s;
}
```

Although a lot of values remain the same, there are two customizations that are part of the *request override*:

- The request method is `HEAD` instead of `GET`.
- We're using the custom `Varnish Health Probe` *User-Agent*.

Assigning the probe to a backend

Once your *probe* is set up and configured, you need to assign it to a backend.

It's a matter of setting the `.probe` property in your *backend* to the name of the probe, as you can see in the example below:

```
vcl 4.1;

probe healthcheck {
    .url = "/health";
    .interval = 10s;
    .timeout = 5s;
}

backend default {
    .host = "127.0.0.1";
    .port = "8080";
    .probe = healthcheck;
}
```

By defining your *probe* as a separate data structure, it can be reused when multiple backends are in use.

The verbose approach is to define the `.probe` property inline, as illustrated in the example below:

```
vcl 4.1;

backend default {
    .host = "127.0.0.1";
    .port = "8080";
    .probe = {
        .url = "/health";
      .interval = 10s;
        .timeout = 5s;
    }
}
```

TCP-only probes

Probes usually perform HTTP requests to check the health of a backend. By using TCP-only probes, the health of a backend is checked by the availability of the TCP connection.

This can be used to probe non-HTTP endpoints. However, TCP-only probes cannot be used with `.url`, `.request`, or `.expected_response` properties.

Here's how you define such a probe:

```
probe tcp_healtcheck {
  .tcponly = 1;
}
```

Keep in mind that TCP-only probes are only available in *Varnish Enterprise*.

UNIX domain sockets

The `backend` data structure has additional properties that can be set with regard to the endpoint it is connecting to.

If you want to connect to your backend using a *UNIX domain socket*, you'll use the `.path` property. It is mutually exclusive with the `.host` property and is only available when you use the `vcl 4.1;` version declaration.

Here's an example of a *UDS-based backend definition*:

```
backend default {
    .path = "/var/run/some-backend.sock";
}
```

Overriding the host header

If for some reason the `Host` header is not set in your *HTTP requests*, you can use the `.host_header` property to override it.

Here's an example:

```
backend default {
    .host = "127.0.0.1";
    .port = "8080";
    .host_header = "example.com";
}
```

This `.host_header` property will be used for both *regular backend requests* and *health probe checks*.

4.4.12 Access control lists

An *access control list (ACL)* is a *VCL* data structure that contains hostnames, IP addresses, and subnets. An *ACL* is used to match *client addresses* and restrict access to certain resources.

Here's how you define an *ACL*:

```
acl admin {
    "localhost";
    "secure.my-server.com";
    "192.168.0.0/24";
    ! "192.168.0.25";
}
```

This *ACL* named `admin` contains the following rules:

- Access from `localhost` is allowed.
- Access from the hostname `secure.my-server.com` is also allowed.

- All IP address in the `192.168.0.0/24` subnet are allowed.
- The only IP address from that range that is not allowed is `192.168.0.25`.

In your *VCL code*, you can then match the *client IP address* to that list, as you'll see in the next example:

```
acl admin {
    "localhost";
    "secure.my-server.com";
    "192.168.0.0/24";
    ! "192.168.0.25";
}

sub vcl_recv {
    if(req.url ~ "^/admin/?" && client.ip !~ admin) {
        return(synth(403,"Forbidden"));
    }
}
```

In this example, we're hooking into `vcl_recv` to intercept requests for `/admin` or any subordinate resource of `/admin/`. If users try to access this resource, we check if their *client IP address* is matched by `acl admin`.

If it doesn't match, an `HTTP 403 Forbidden` error is returned synthetically.

4.4.13 Functions

Complex logic in a programming language is usually abstracted away by functions. This is also the case in *VCL*, which has a number of native functions.

The number of functions is limited, but extra functions are available in the wide range of *VMODs* that are supported by *Varnish*.

In *chapter 5*, we'll talk about *VMODs* and how their functions extend the capabilities of *Varnish*.

ban()

`ban()` is a function that adds an expression to the *ban list*. These expressions are matched to cached objects. Every matching object is then removed from the cache.

In essence, the `ban()` function exists to invalidate multiple objects at the same time.

Although *banning* will be covered in detail in *chapter 5*, here's a quick example:

```
ban("obj.age > 1h");
```

Multiple expressions can be chained using the `&&` operator.

hash_data()

The `hash_data()` function is used within the `vcl_hash` subroutine and is used to append *string data* to the *hash input* that is used to lookup an object in cache.

Let's just revisit the *built-in VCL* for `vcl_hash` where `hash_data()` is used:

```
sub vcl_hash {
    hash_data(req.url);
    if (req.http.host) {
        hash_data(req.http.host);
    } else {
        hash_data(server.ip);
    }
    return (lookup);
}
```

synthetic()

The `synthetic()` function prepares a *synthetic response body* and uses a *string argument* for its input. This function can be used within `vcl_synth` and `vcl_backend_error`.

Here's an example for `vcl_synth`:

```
synthetic(resp.reason);
```

However, this function is no longer used in the *built-in VCL*. As of *Varnish Cache 5.0*, it is recommended to instead use `set beresp.body = {""};`.

regsub()

The `regsub()` function is a very popular function in *Varnish*. This function performs *string substitution using regular expressions*. Basically, do *find/replace on the first occurrence* using a *regex pattern*.

This is the *API* of this function:

```
regsub(string, regex, sub)
```

- The `string` argument is your input.
- The `regex` argument is the *regular expression* you're using to match what you're looking for in the *input string*.
- The `sub` argument is what the *input string* will be substituted with.

A practical example

Here's a really practical example where we use `regsub()` to extract a *cookie value*:

```
vcl 4.1;

sub vcl_hash {
    hash_data(regsub(req.http.Cookie,"(;|^)language=([a-z]{2})
(;|$)","\2"));
}
```

Let's break it down because it looks quite complex.

This `vcl_hash` subroutine is used to extend the *built-in VCL* and to add the value of the `language` cookie to the hash. This creates a cache variation per language.

We really don't want to hash the entire cookie because that will drive our hit rate down, especially when there are tracking cookies in place.

In order to extract the exact cookie value we need, we'll match the `req.http.Cookie` header to a *regular expression* that uses grouping. In the substitution part, we can refer to those groups to extract the value we want.

Here's the regular expression:

```
(;|^)language=([a-z]{2})(;|$)
```

This regular expression looks for a `language=` occurrence, followed by *two letters*. These letters represent the language. This language cookie can occur at the beginning of the cookie string, in the middle, or at the end. The `(;|^)` and `(;|$)` statements ensure that this is possible.

Because we're using parentheses for grouping, the group where we match the language itself, is indexed as *group two*. This means we can refer to it in the `regsub()` function as `\2`.

So if we look at the entire `regsub()` example:

```
regsub(req.http.Cookie,"(;|^)language=([a-z]{2})(;|$)","\2")
```

And let's imagine this is our `Cookie` header:

```
Cookie: privacy_accepted=1;language=en;sessionid=03F1C5944FF4
```

Given the *regular expression* and the *group referencing*, the output of this `regsub()` function would be `en`.

This means that `en` will be added to the hash along with the *URL* and the *host header*.

When the `Cookie` header doesn't contain a `language` cookie, an empty string is returned. When there is no `Cookie` header, an empty string is returned as well. This means we don't risk *hash-key collisions* when the cookie isn't set. #### regsuball()

The `regsuball()` function is very similar to the `regsub()` function we just covered. The only difference is where `regsub()` matches and replaces the first occurrence of the pattern, `regsuball()` matches all occurrences.

Even the *function API* is identical:

```
regsuball(string, regex, sub)
```

- The `string` argument is your input.
- The `regex` argument is the *regular expression* you're using to match what you're looking for in the *input string*.
- The `sub` argument is what the *input string* will be substituted with.

A practical example

Let's have a look at a similar example, where we'll strip off some cookies again. Instead of matching the values we want to keep, we'll match the values we want to remove. We need to ensure that all occurrences are matched, not just the first occurrence. That's why we use `regsuball()` instead of `regsub()`:

```
regsuball(req.http.Cookie,"_g[a-z0-9_]+=[^;]*($|;\s*)","")
```

What this example does, is remove all *Google Analytics* cookies. This is the list of cookies we need to remove:

- `_ga`
- `_gid`
- `_gat`
- `_gac_<property-id>`

Instead of stripping them off one by one, we can use the `_g[a-z0-9_]+=[^;]*($|;\s*)` regular expression to match them all at once. In the end we'll replace the matched cookies with an empty string.

This could be the raw value of your `req.http.Cookie` header:

```
cookie1=a; _ga=GA1.2.1915485056.1587105100;cookie2=b; _gid=-
GA1.2.873028102.1599741176; _gat=1
```

And the end result is the following:

```
cookie1=a;cookie2=b
```

4.4.14 Subroutines

At this point, the term *subroutine* in a *VCL* context is hopefully not a foreign concept. We've been through the *Varnish finite state machine* multiple times, you've seen the corresponding *built-in VCL* code. But what you might not know is that you can define your own subroutines.

Need an example? Here you go:

```
vcl 4.1;

sub skipadmin {
    if(req.url ~ "^/admin/?") {
        return(pass);
    }
}
```

```
sub vcl_recv {
    call skipadmin;
}
```

The `skipadmin` subroutine is entirely custom and is called within `vcl_recv` using the `call` statement. The purpose of custom subroutines is to allow code to be properly structured and functionality compartmentalized.

The example above groups the logic to bypass requests to the *admin panel* in a separate subroutine, which is then called from within `vcl_recv`.

> You are free to name your custom subroutine whatever you want, but keep in mind that the `vcl_` naming prefixes are reserved for the *Varnish finite state machine*. Please also keep in mind that *a subroutine is not a function:* it does not accept input parameters, and it doesn't return values. It's just a procedure that is called.

4.4.15 Include

Not all of your *VCL* logic should necessarily be in the same *VCL file*. When the line count of your *VCL file* increases, readability can become an issue.

To tackle this issue, *VCL* allows you to *include* VCL from other files. The *include* syntax is not restricted to subroutines and fixed language structures, even individual lines of *VCL code* can be included.

The `include "<filename>;"` syntax will tell the compiler to read the file and *copy* its contents into the main *VCL* file.

When including a file, the order of execution in the main *VCL* file will be determined by the order of inclusion.

This means that each include can define its own VCL routing logic and if an included file exits the subroutine early, it will bypass any logic that followed that return statement.

The *built-in VCL* follows this logic and can be thought of as an included file at the end of your *VCL*. This means that if you put a `return` statement anywhere in your *VCL*, the *built-in VCL* logic will be skipped since it is always appended at the end of your *VCL*.

So let's talk about the previous example, where the `skipadmin` subroutine is used and put the *custom subroutine* in a separate file:

```
#This is skipadmin.vcl

sub skipadmin {
    if(req.url ~ "^/admin/?") {
        return(pass);
    }
}
```

In your main *VCL file*, you'll use the `include` syntax to include `skipadmin.vcl`:

```
vcl 4.1;

include "skipadmin.vcl";

sub vcl_recv {
    call skipadmin;
}
```

And the resulting compiled *VCL* would be:

```
vcl 4.1;

sub skipadmin {
    if(req.url ~ "^/admin/?") {
        return(pass);
    }
}

sub vcl_recv {
    call skipadmin;
}
```

4.4.16 Import

The `import` statement can be used to import *VMODs*. These are *Varnish modules*, written in *C-code*, that are loaded into *Varnish* and offer a *VCL interface*. These modules basically enrich the *VCL syntax* without being part of the Varnish core.

We'll cover all the *ins and outs* of *VMODs* in the next chapter.

Here's a quick example:

```
vcl 4.1;

import std;

sub vcl_recv {
    set req.url = std.querysort(req.url);
}
```

This example uses `import std;` to import *Varnish's standard library* containing a set of *utility functions*. The `std.querysort()` function will alphabetically sort the *query string parameters* of a URL, which has a beneficial impact on the hit rate of the cache.

4.5 VCL objects and variables

VCL, the *finite state machine*, *hooks* and *subroutines*: you know what it does and how it works by now. Throughout this chapter, you've seen quite a number of *VCL examples* that override the behavior of *Varnish* by checking or changing the value of a *VCL variable*.

In this section, we'll give you an overview of what is out there in terms of *VCL variables* and what *VCL objects* they belong to.

You can group the variables as follows:

- *Connection*-related variables. Part of the `local`, `server`, `remote` and `client` objects
- *Request*-related variables. Part of the `req` and `req_top` objects
- *Backend request*-related variables. Part of the `bereq` object
- *Backend response*-related variables. Part of the `beresp` object
- *Cache object*-related variables. Part of the `obj` object
- *Response*-related variables. Part of the `resp` object
- *Session*-related variables. Part of the `sess` object
- *Storage*-related variables. Part of the `storage` object

This book is not strictly reference material. Although there is a lot of educational value, the main purpose is to inspire you and show what *Varnish* is capable of. That's why we will not list all *VCL variables*: we'll pick a couple of useful ones, and direct you to the rest of them, which can be found at http://varnish-cache.org/docs/6.0/reference/vcl.html#vcl-variables.

4.5.1 Connection variables

There are four *connection*-related objects available in *VCL*, and their meaning depends on your topology:

- `client`: the client that sends the *HTTP request*
- `server`: the server that receives the *HTTP request*
- `remote`: the remote end of the *TCP connection* in *Varnish*
- `local`: the local end of the *TCP connection* in *Varnish*

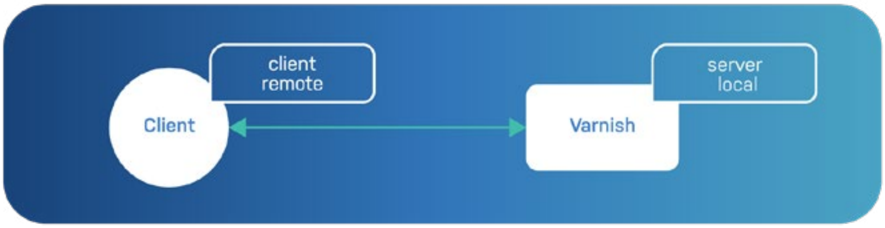
PROXY vs no PROXY

PROXY protocol connections were introduced in *Varnish 4.1*. The main purpose is to allow a proxy node to provide the connection information of the originating client requesting the connection.

In a setup where *Varnish* is not using the *PROXY protocol*, the `client` and `remote` objects refer to the same connection: the client is the remote part of the TCP connection.

The same applies to `server` and `local`: the server is the local part of the TCP connection.

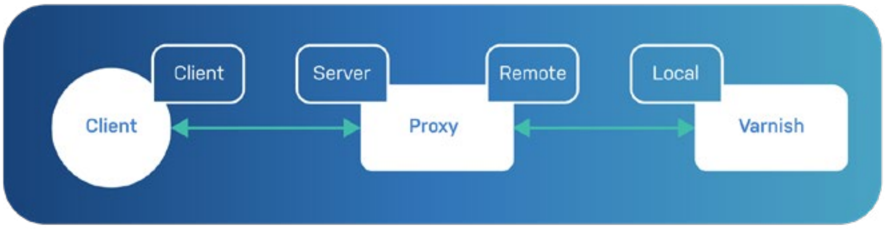
The following diagram illustrates this:



Connections without PROXY

A variable like `client.ip` will be used to get the IP address of the client. But as explained, the value of `remote.ip` will be identical. And `server.ip` will match the `local.ip` value.

But when the *PROXY protocol* is used, there is an extra hop in front of *Varnish*. This extra node communicates with *Varnish* over the *PROXY protocol*. In that kind of setup, the variables each have their own meaning, as you can see in the diagram below:



Connections with PROXY

`client.ip` is populated by the information of the *PROXY protocol*. It contains the IP address of the *original client*, regardless of the number of hops that were used in the process.

`server.ip` is also retrieved from the *PROXY protocol*. This variable represents the IP address of the server to which the client connected. This may be a node that sits many hops in front of *Varnish*.

`remote.ip` is the IP address of the node that sits right in front of *Varnish*.

`local.ip` is *Varnish*'s IP address.

The IP type

For values like `client.ip` and other variables that return the `IP` type, they are more than a string containing the IP address. These types also contain the *port* that was used by the connection.

The way you can extract the *integer value of the port* is by using the `std.port()` function that is part of `vmod_std`.

Here's an example:

```
vcl 4.1;

import std;

sub vcl_recv {
    if(std.port(server.ip) == 443) {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

This example will extract the *port value* from `server.ip`. If the value equals `443`, it means the *HTTPS* port was used and the `X-Forwarded-Proto` header should be set to `https`. Otherwise, the value is `http`.

When connections are made over *UNIX domain sockets*, the IP value will be `0.0.0.0`, and port value will be `0`.

Local variables

The `local` object has two interesting variables:

- `local.endpoint`
- `local.socket`

Both of these variables are only available using the `vcl 4.1` syntax.

`local.endpoint` contains the *socket address* for the `-a` socket.

If `-a http=:80` was set in `varnishd`, the `local.endpoint` value would be `:80`.

Whereas `local.endpoint` takes the *socket value*, `local.socket` will take the *socket name*. If we take the example where `-a http=:80` is set, the value for `local.socket` will be `http`.

If, like many people, you don't name your sockets, *Varnish will do so for you*.

The naming pattern that *Varnish* uses for this is `a:%d`. So the name of the first socket will be `a0`.

Identities

Both the client and the server object have an `identity` variable that identifies them.

`client.identity` identifies the client, and its default value is the same value as `client.ip`. This is a *string* value, so you can assign what you like.

This variable was originally used by the *client director* for client-based load balancing. However, the last version of *Varnish* that supported this was *version 3*.

Naturally, the `server.identity` variable will identify your server. If you specified `-i` as a runtime parameter in `varnishd`, this will be the value. If you run multiple *Varnish instances* on a single machine, this is quite convenient.

But by default, `server.identity` contains the hostname of your server, which is the same value as `server.hostname`.

4.5.2 Request variables

The `req` object allows you to inspect and manipulate incoming *HTTP requests*.

The most common variable is `req.url`, which contains the *request URL*. But there's also `req.method` to get the *request method*. And every *request header* is accessible through `req.http.*`.

A request example

Imagine the following *HTTP request*:

```
GET / HTTP/1.1
Host: localhost
User-Agent: curl
Accept: */*
```

Varnish will populate the following *request variables*:

- `req.method` will be `GET`.
- `req.url` will be `/`.
- `req.proto` will be `HTTP/1.1`.
- `req.http.host` will be `localhost`.
- `req.http.user-agent` will be `curl`.
- `req.http.accept` will be `*/*`.
- `req.can_gzip` will be `false` because the request didn't contain an `Accept-Encoding: gzip` header.

There are also some internal variables that are generated when a request is received:

`req.hash` will be `3k0f0yRKtKt7akzkyNsTGSDOJAZOQowTwKWhu5+kIu0=` if we *base64 encode* the *blob* value.

> `req.hash` is the hash that *Varnish* will use to identify the object in cache upon lookup. `vmod_blob` is needed to convert `req.hash` into a readable `base64` representation.

`req.is_hitmiss` and `req.is_hitmiss` will be `true` if there's a *uncacheable object* stored in cache from a previous request.

If that's the case, the *waiting list* will be bypassed. Otherwise this value is `false`.

`req.esi_level` will be `0` because this request is not an internal subrequest. It was triggered by an actual client.

Top-level requests and Edge Side Includes

Edge Side Includes (ESI) was introduced in *chapter 3* and there are some *VCL variables* in place to facilitate *ESI*.

At the request level, there is a `req.esi_level` variable to check the level of depth and a `req.esi` variable to toggle *ESI* support.

Parsing *ESI* actually happens at the backend level and is done through `beresp.do_esi`, but we'll discuss that later on when we reach *backend request variables*.

An *ESI* request triggers an internal subrequest in *Varnish*, which increments the `req.esi_level` counter.

When you're in an *ESI* subrequest, there is also some context available about the *top-level request* that initiated the subrequest. The `req_top` object provides that context.

- `req_top.url` returns the URL of the parent request.
- `req_top.http.*` contains the request headers of the parent request.
- `req_top.method` returns the request method of the parent request.
- `req_top.proto` returns the protocol of the parent request.

If `req_top` is used in a *non-ESI context*, their values will be identical to the `req` object.

If you want to know whether or not the top-level request was requesting the homepage, you could use the following *example VCL code*:

```
vcl 4.1;
import std;

sub vcl_recv {
    if(req.esi_level > 0 && req_top.url == "/") {
        std.log("ESI call for the homepage");
    }
}
```

This example will log `ESI call for the homepage` to *Varnish's Shared Memory Log*, which we'll cover in *chapter 7*.

4.5.3 Backend request variables

Backend requests are the requests that Varnish sends to the origin server when an object cannot be served from cache. The `bereq` contains the necessary backend request information and is built from the `req` object.

In terms of scope, the `req` object is accessible in *client-side subroutines*, whereas the `bereq` object is only accessible in *backend subroutines*.

Although both objects are quite similar, they are not identical. The backend requests do not contain the *per-hop fields* such as the `Connection` header and the `Range` header.

All in all, the `bereq` variables will look quite familiar:

- `bereq.url` is the backend request URL.
- `bereq.method` is the backend request method.
- `bereq.http.*` contains the backend request headers.
- `bereq.proto` is the backend request protocol that was used.

On the one hand, `bereq` provides a copy of the *client request information* in a *backend context*. But on the other hand, because the backend request was initiated by *Varnish*, we have a lot more information in `bereq`.

Allow us to illustrate this:

`bereq.connect_timeout`, `bereq.first_byte_timeout`, and `bereq.between_bytes_timeout` contain the timeouts that are applied to the backend.

They were either set in the `backend` or are the default values from the `connect_timeout`, `first_byte_timeout`, and `between_bytes_timeout` runtime parameters.

`bereq.is_bgfetch` is a *boolean* that indicates whether or not the backend request is made in the background. When this variable is `true`, this means *the client hit an object in grace*, and a new copy is fetched in the background.

`bereq.backend` contains the backend that Varnish will attempt to fetch from. When it is used in a string context, we just get the backend name.

4.5.3 Backend response variables

Where there's a *backend request*, there is also a *backend response*. Again, this implies that an object couldn't be served from cache.

The `beresp` object contains all the relevant information regarding the backend response.

- `beresp.proto` contains the protocol that was used for the backend response.
- `beresp.status` is the *HTTP status code* that was returned by the origin.
- `beresp.reason` is the *HTTP status message* that was returned by the origin.
- `beresp.body` contains the *response body*, which can be modified for synthetic responses.
- `beresp.http.*` contains all response headers.

VFP-related backend response variables

There are also a bunch of *backend response variables* that are related to the *Varnish Fetch Processors (VFP)*. These are booleans that allow you to *toggle* certain features:

- `beresp.do_esi` can be used to enable *ESI parsing*.
- `beresp.do_stream` can be used to disable *HTTP streaming*.
- `beresp.do_gzip` can be used to explicitly compress *non-gzip* content.
- `beresp.do_gunzip` can be used to explicitly uncompress *gzip* content and store the *plain text* version in cache.

Timing-related backend response variables

- `beresp.ttl` contains the objects *remaining time to live (TTL)* in seconds.
- `beresp.age` (read-only) contains the age of an object in seconds.
- `beresp.grace` is used to set the *grace period* of an object.
- `beresp.keep` is used to keep *expired and out of grace* objects around for conditional requests.

These variables return a *duration type*. `beresp.age` is read-only, but all the others can be set in the `vcl_backend_response` or `vcl_backend_error` subroutines.

Other backend response variables

- `beresp.was_304` indicates whether or not our *conditional fetch* got an *HTTP 304* response before being turned into an *HTTP 200*.
- `beresp.uncacheable` is inherited from `bereq.uncacheable` and is used to flag objects as uncacheable. This results in a `hit-for-miss`, or a `hit-for-pass` object being created.
- `beresp.backend` returns the backend that was used.

`beresp.backend` returns a `backend` object. You can then use `beresp.backend.name` and `beresp.backend.ip` to get the name and IP address of the backend that was used.

4.5.4 Object variables

The term *object* refers to what is stored in cache. It's read-only and is exposed in *VCL* via the `obj` object.

Here are some `obj` variables:

- `obj.proto` contains the *HTTP protocol version* that was used.
- `obj.status` stores the *HTTP status code* that was used in the response.
- `obj.reason` contains the *HTTP reason phrase* from the response.
- `obj.hits` is a *hit counter*. If the counter is `0` by the time `vcl_deliver` is reached, we're dealing with a *cache miss*.
- `obj.http.*` contains all *HTTP headers* that originated from the HTTP response.
- `obj.ttl` is the object's remaining *time to live* in seconds.
- `obj.age` is the objects age in seconds.
- `obj.grace` is the grace period of the object in seconds.
- `obj.keep` is the amount of time an *expired and out of grace* object will be kept in cache for conditional requests.
- `obj.uncacheable` determines whether or not the cached object is *uncacheable*.

As explained earlier, even non-cacheable objects are kept in cache and are marked *uncacheable*. By default these objects perform `hit-for-miss` logic, but can also be configured to perform `hit-for-pass` logic.

For the sake of efficiency, uncacheable objects are a lot smaller in size and don't contain all the typical response information.

4.5.5 Response variables

In *HTTP*, and in *Varnish* too for that matter, there is always a request and always a response. The `resp` object contains the necessary information about the response that is going to be returned to the client.

In case of a *cache hit* the `resp` object is populated from the `obj` object. For a *cache miss* or if the cache was bypassed, the `resp` object is populated by the `beresp` object.

When a *synthetic response* is created, the `resp` object is populated from synth.

And again, the `resp` variables will look very familiar:

- `resp.proto` contains the protocol that was used to generate the *HTTP response*.
- `resp.status` is the *HTTP status code* for the response.
- `resp.reason` is the *HTTP reason phrase* for the response.
- `resp.http.*` contains the *HTTP response headers* for the response.
- `resp.is_streaming` indicates whether or not the response is being streamed while being fetched from the backend.
- `resp.body` can be used to produce a *synthetic response body* in `vcl_synth`.

4.5.1 Storage variables

In `varnishd` you can specify one or more *storage backends*, or *stevedores* as we call them. The `-s` runtime parameter is used to indicate where objects will be stored.

Here's a typical example:

```
varnishd -a :80 -f /etc/varnish/default.vcl -s malloc,256M
```

This *Varnish* instance will store its objects in memory and will allocate a maximum amount of *256 MB*.

In *VCL* you can use the `storage` object to retrieve the *free space* and the *used space* of *stevedore*.

In this case you'd use `storage.s0.free_space` to get the free space, and `storage.s0.used_space` to get the used space.

When a *stevedore* is unnamed, *Varnish* uses the `s%d` naming scheme. In our case the *stevedore* is named `s0`.

Let's throw in an example of a *named stevedore*:

```
varnishd -a :80 -f /etc/varnish/default.vcl -s memory=malloc,256M
```

To get the *free space* and *used space*, you'll have to use `storage.memory.free_space` and `storage.memory.used_space`.

4.6 Making changes

In the previous section of the book, we took a *deep dive* into all the *VCL variables*. We also covered the *built-in VCL* and the *Varnish finite state machine* extensively.

In this section, we'll cover some basic scenarios on how to make meaningful changes in your *VCL*.

4.6.1 Excluding URL patterns

You want to cache as much as possible, but in reality you can't: resources that are *stateful* are often hard or impossible to cache.

When caching a *stateful resource* would result in too many variations, it's not worth caching.

A very common pattern in *VCL* is to exclude *URL patterns* and do a `return(pass)` when they are matched.

This one comes right out of the *Magento 2 VCL file*:

```
vcl 4.1;

sub vcl_recv {
    if (req.url ~ "/checkout") {
        return (pass);
    }
}
```

Because the `/checkout` URL namespace is a very personalized experience, it's not really cacheable: you're dealing with logins and payment details. You really have to *pass* here.

And here's another example coming from *WordPress*:

```
vcl 4.1;

sub vcl_recv {
    if (req.url ~ "wp-(login|admin)" || req.url ~ "preview=true") {
        return (pass);
    }
}
```

If the URL starts with `/wp-login` or `/wp-admin`, you're trying to access the admin panel, which is not cacheable.

This is also the case when you're previewing cacheable pages when being logged in. As a result, pages containing `preview=true` in the URL won't be cached either.

> Notice that only slight modifications were required to achieve our goal. Because we only `return(pass)` for specific patterns, the rest of the application can still rely on the *built-in VCL*. As always, the *built-in VCL* is your *safety net*.

4.6.2 Sanitizing the URL

Cache objects are identified by the URL. The URL is not just the identifier of the resource, it can also contain *query string parameters*. But in terms of *hashing*, *Varnish* treats the URL as string.

This means that the slightest change in any of the *query string parameters* will result in a new hash, which in its turn results in a *cache miss*.

There are some strategies where the *URL* is sanitized in order to avoid too many cache misses.

Here's some *VCL* to sanitize your URL:

```
vcl 4.1;

import std;

sub vcl_recv {
    # Sort the query string parameters alphabetically
    set req.url = std.querysort(req.url);

    # Remove third-party tracking parameters
    if (req.url ~ "(\?|&)(utm_source|utm_medium|utm_campaign|utm_con-
tent)=") {
        set req.url = regsuball(req.url, "&(utm_source|utm_medi-
um|utm_campaign|utm_content)=([A-z0-9_\-\.%25]+)", "");
        set req.url = regsuball(req.url, "\?(utm_source|utm_medi-
um|utm_campaign|utm_content)=([A-z0-9_\-\.%25]+)", "?");
        set req.url = regsub(req.url, "\?&", "?");
        set req.url = regsub(req.url, "\?$", "");
    }

    # Remove hashes from the URL
    if (req.url ~ "\#") {
    set req.url = regsub(req.url, "\#.*$", "");
    }

    # Strip off trailing question marks
    if (req.url ~ "\?$") {
    set req.url = regsub(req.url, "\?$", "");
    }
}
```

Alphabetic sorting

The first step is to *sort the query string parameters alphabetically*. If you change the order of a *query string parameter*, you change the string, which results in a *cache miss*.

The `std.querysort` function from `vmod_std` does this for you. It's a simple modification that can have a massive impact.

Removing tracking query string parameters

Marketing people are keen to figure out how their campaigns are performing. *Google Analytics* can add campaign context to URL by adding *tracking URL parameters*.

Here's a list of these parameters:

- `utm_source`
- `utm_medium`
- `utm_campaign`
- `utm_content`

In the example above we're stripping them off because they are meaningless to the server, and they mess with our hit rate. Because these parameters are processed *client-side*, removing them `server-side` has no negative impact.

The `regsub()` and `regsuball()` functions in the example above strip off unwanted *tracking query string parameters* using regular expressions.

Removing URL hashes

In *HTML*, we can mark page sections using *anchors*, as illustrated below:

```
<a name="my-section"></a>
```

You can directly scroll to this section by adding a *hash* to the URL. Here's how that looks:

```
http://example.com/#my-section
```

We've said it 100 times at least, and we'll have to repeat it again: changing the URL changes the *lookup hash* for the cache. These *URL hashes* are also meaningless in a *server-side context* and also mess with our hit rate.

Your best move is to strip them off. The `set req.url = regsub(req.url, "\#.*$", "");` does this for you.

Removing trailing question marks

In the same vein as the previous example, we want to avoid *cache misses* by stripping off *trailing question marks*.

The `?` in a *URL* indicates the start of the query string parameters. But if the question mark is at the end of the URL, there aren't any parameters, so we need to strip off the `?`. This is done by `set req.url = regsub(req.url, "\?$", "");`

4.6.3 Stripping off cookies

Cookies are indicators of *state*. And *stateful content* should not be cached unless the variations are manageable.

But a lot of *cookies* are there to personalize the experience. They keep track of *session identifiers*, and there are also tracking cookies that change upon every request.

There are two approaches to get rid of them:

- Identify the cookies you want to remove
- Identify the cookies you want to keep

Removing select cookies

```
vcl 4.1;

    sub vcl_recv {
    # Some generic cookie manipulation, useful for all templates that
follow
    # Remove the "has_js" cookie
    set req.http.Cookie = regsuball(req.http.Cookie, "has_js=[^;]+(;
)?", "");

    # Remove any Google Analytics based cookies
    set req.http.Cookie = regsuball(req.http.Cookie, "__utm.=[^;]+(;
)?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "_ga=[^;]+(;
)?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "_gat=[^;]+(;
)?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "utmctr=[^;]+(;
)?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "utmcmd.=[^;]+(;
)?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "utmccn.=[^;]+(;
)?", "");

    # Remove DoubleClick offensive cookies
    set req.http.Cookie = regsuball(req.http.Cookie, "__gads=[^;]+(;
)?", "");

    # Remove the Quant Capital cookies (added by some plugin, all __
qca)
    set req.http.Cookie = regsuball(req.http.Cookie, "__qc.=[^;]+(;
)?", "");

    # Remove the AddThis cookies
    set req.http.Cookie = regsuball(req.http.Cookie, "__atuv.=[^;]+(;
)?", "");

    # Remove a ";" prefix in the cookie if present
    set req.http.Cookie = regsuball(req.http.Cookie, "^;\s*", "");

    # Are there cookies left with only spaces or that are empty?
    if (req.http.cookie ~ "^\s*$") {
        unset req.http.cookie;
    }
}
```

This *VCL snippet* will identify every single cookie pattern that needs to be removed. It ranges from *Google Analytics* tracking cookies, to *DoubleClick*, all the way to *AddThis*.

Every cookie that matches is removed. If you end up with nothing more than a set of *whitespace characters*, this means there weren't any cookies left, and we remove the entire `Cookie` header.

Cookies that weren't removed will remain in the `Cookie` header and will fall back on the *built-in VCL*, which will perform a `return(pass);`. This is not really a problem because it's by design.

Removing all but some cookies

The opposite is actually a lot easier: only keep a couple of cookies, and remove the rest.

Here's an example that does that:

```
vcl 4.1;

sub vcl_recv {
    if (req.http.Cookie) {
    set req.http.Cookie = ";" + req.http.Cookie;
    set req.http.Cookie = regsuball(req.http.Cookie, "; +", ";");
    set req.http.Cookie = regsuball(req.http.Cookie, ";(PHPSESSID)=",
"; \1=");
    set req.http.Cookie = regsuball(req.http.Cookie, ";[^ ][^;]*",
"");
    set req.http.Cookie = regsuball(req.http.Cookie, "^[; ]+|[; ]+$",
"");

    if (req.http.cookie ~ "^\s*$") {
        unset req.http.cookie;
    }
}
```

Imagine having a *PHP* web application that has an *admin panel*. When you create a *session* in *PHP*, the `PHPSESSID` cookie is used by default. This is the only cookie that matters *server-side* in our application.

When this cookie is set, you're logged in, and the page can no longer be cached. This example looks quite complicated, but it just sets up a cookie format where `PHPSESSID` can easily be identified, and other cookies are replaced with an empty string.

And again: if you end up with a collection of whitespace characters, you can just remove that cookie.

Using vmod_cookie

If you're on *Varnish Cache 6.4* or later, `vmod_cookie` is shipped by default.

Here's the first example, where we explicitly remove cookies using `vmod_cookie`:

```
vcl 4.1;

import cookie;

sub vcl_recv {
    cookie.parse(req.http.cookie);
    cookie.filter("_ga,_gat,utmctr,__gads,has_js");
    cookie.filter_re("(__utm.|utmcmd.|utmccn.|__qc.|__atuv.)");
    set req.http.cookie = cookie.get_string();
    if (req.http.cookie ~ "^\s*$") {
         unset req.http.cookie;
     }
}
```

Here's the second example, where we only keep the `PHPSESSID` cookie using `vmod_cookie`:

```
vcl 4.1;

import cookie;

sub vcl_recv {
    cookie.parse(req.http.cookie);
    cookie.keep("PHPSESSID");
    set req.http.cookie = cookie.get_string();
    if (req.http.cookie ~ "^\s*$") {
         unset req.http.cookie;
     }
}
```

You have to admit, this is a lot simpler. There are still regular expressions involved, but only to match cookie names. The complicated logic to match names, values, and separators is completely abstracted.

Using vmod_cookieplus

If you're not on *Varnish Cache 6.4*, you can still benefit from another *cookie VMOD*: *Varnish Enterprise* ships `vmod_cookieplus`. This is an Enterprise version of `vmod_cookie` that has more features and a slightly different API.

Here's the first example, where we explicitly remove cookies using `vmod_cookieplus`:

```
vcl 4.1;

import cookieplus;

sub vcl_recv {
    cookieplus.delete("_ga");
    cookieplus.delete("_gat");
    cookieplus.delete("utmctr");
    cookieplus.delete("__gads");
    cookieplus.delete("has_js");
    cookieplus.delete_regex("(__utm.|utmcmd.|utmccn.|__qc.|__
atuv.)");
    cookieplus.write();
}
```

And here's how we only keep the `PHPSESSID` cookie using `vmod_cookieplus`:

```
vcl 4.1;

import cookieplus;

sub vcl_recv {
    cookieplus.keep("PHPSESSID");
    cookieplus.write();
}
```

As you can see, `vmod_cookieplus` doesn't need to be initialized, the `Cookie` header doesn't need to be parsed in advanced, and although there is a `cookieplus.write()` function, it doesn't require writing the value back to `req.http.Cookie`.

A final note about `vmod_cookieplus` is that the deletion process doesn't leave you with an empty `Cookie` header, unlike `vmod_cookie`. If the *cookie* is empty in the end, it is stripped off automatically.

4.6.4 Sanitizing content negotiation headers

We already covered this in *chapter 3*, but sanitizing your *content negotiation headers* is important, especially if you're planning on *varying* on them.

By *content negotiation headers* we mean `Accept` and `Accept-Language`. There's also the `Accept-Encoding` header, but *Varnish* handles this one out of the box.

The `Accept` request header defines what *content types* the client supports. This could be `text/plain`, `text/html`, or even `application/json`.

The `Accept-Language` request header defines what *languages* the client understands. This is an ideal way to serve *multilingual content* with explicit language selection.

The problem with these headers is that they can have so many variations. If you would do a `Vary: Accept-Language` your hit rate might drop massively.

It's not only the vast number of languages that cause this, but also the order, the priority and the localization of these languages.

You probably have a pretty good idea which languages your web platform supports. Just allow them and rely on a default value when the client's preferred language is not supported.

Here's the example we used in *chapter 3*:

```
vcl 4.1;

import accept;

sub vcl_init {
    new lang = accept.rule("en");
    lang.add("nl");
}

sub vcl_recv {
    set req.http.Accept-Language = lang.filter(req.http.Accept-Lan-
guage);
}
```

This is the `Accept-Language` header in my browser:

```
Accept-Language: nl-BE,nl;q=0.9,en-US;q=0.8,en;q=0.7
```

These settings are personalized, and your browser settings will undoubtedly differ. Without a proper cleanup, it is impossible to get a decent hit rate when you *vary* on this header.

My *VCL script* will pick `nl` as the selected language. If `nl` is nowhere to be found in the `Accept-Language` header, `en` will be the fallback.

`vmod_accept` also works for the `Accept` header. Here's an example:

```
vcl 4.1;

import accept;

sub vcl_init {
    new format = accept.rule("text/plain");
    format.add("text/html");
    format.add("application/json");
}

sub vcl_recv {
    set req.http.accept = format.filter(req.http.accept);
}
```

In this example we support content that is *HTML* or *JSON*. Anything else will result in the `text/plain` *MIME type*, which just means the document is not parsed and returned as *plain text*.

By sanitizing your *content negotiation headers*, you limit the variations per header, and you can safely issue a `Vary: Accept`, or a `Vary: Accept-Language` in your web application.

4.6.5 Overriding TTLs

Developer empowerment is a term we use a lot when we talk about caching. In *chapter 3*, we covered it in great detail: *HTTP* has so many built-in mechanisms to improve the cacheability of your web application. If you use the right headers, you're in control.

However, in the real world, `Cache-Control` and `Expires` headers aren't always used. And quite often you'll find `Cache-Control: private, no-cache, no-store` headers on a perfectly cacheable page.

Refactoring your code and implementing the proper *HTTP headers* is a good idea. But every now and then, you'll run into a *legacy application* that you wouldn't want to touch with a stick: "it works, but don't ask how".

That's where *VCL* comes into play. The `beresp.ttl` value is determined by the value of `Cache-Control` or `Expires`. But you can override the value if required.

Static data example

The following example will identify *images and videos* based on the `Content-Type` header. For those resources we set the *TTL* to *one year* because it's static data, and it's not supposed to change.

And if the `Cache-Control` header contains `no-cache`, `no-store`, or `private`, we strip off the `Cache-Control` header. Otherwise, the *built-in VCL* would turn this into a `hit-for-miss`:

```
vcl 4.1;

sub vcl_backend_response {
    if (beresp.http.Content-Type ~ "^(image|video)/") {
        if(beresp.http.Cache-Control ~ "(?i:no-cache|no-store|pri-
vate)")){
            unset beresp.http.Cache-Control;
        }
        set beresp.ttl = 1y;
    }
}
```

Overriding the default TTL

Varnish's default TTL is defined by the `default_ttl` runtime parameter. By default this is *120 seconds*.

If you change the value of the `default_ttl` parameter, *Varnish* will use that value if the *HTTP response* doesn't contain a *TTL*.

You can also do it in *VCL*:

```
vcl 4.1;

sub vcl_backend_response {
    set beresp.ttl = 1h;
}
```

Zero TTLs are evil

The lifetime of an object is defined by its *TTL*. If the *TTL* is zero, the object is stale. If *grace* and *keep* values are set, the *TTL* can even be less than zero.

An instinctive reaction is to set `beresp.ttl = 0s` if you want to make sure an object is not stored in cache. However, you're doing more harm than good.

The *built-in VCL* has a mechanism in place to deal with *uncacheable* content:

```
set beresp.ttl = 120s;
set beresp.uncacheable = true;
```

By setting `beresp.uncacheable = true`, we're deciding to cache the decision not to cache, as explained earlier in the book. We call this `hit-for-miss` and `hit-for-pass`, and these objects are kept for *two minutes*.

This metadata is used to bypass the waiting list, as we explained in the *under the hood* section in *chapter 1*.

By setting `beresp.ttl = 0s`, you lose the metadata, requests for this resource are put on the waiting list, and *request coalescing* will not satisfy the request.

The end result is serialization, which means that these items on the waiting list are processed serially rather than in parallel. The impact of serialization is increased latency for the clients.

> We said it before, and we'll say it again: *zero TTLs are evil*

4.6.6 Dealing with websockets

Websockets are a mechanism that offers *full-duplex* communication over a single *TCP* connection. *Websockets* are used for real-time bi-directional communication between a client and a server without the typical request-response exchange.

Websockets are initiated via *HTTP*, but the `Connection: Upgrade` and `Upgrade: websocket` headers will trigger a *protocol upgrade*. This protocol upgrade results in a *persisted open connection* between client and server, where another protocol is used for communication over the *TCP connection*.

Here's an example request:

```
GET /chat
Host: example.com
Origin: https://example.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

This could be the following:

```
HTTP 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

And as soon is the protocol has been switched, we're no longer communicating over *HTTP*.

If you remember the *Varnish finite state machine*, and the various *return statements*, then you'll probably agree that `return(pipe)` is the way to go here.

The `vcl_pipe` subroutine is used to deal with traffic that couldn't be identified as *HTTP*. The *built-in VCL* uses it when *Varnish* notices an unsupported *request method*. The *pipe* we refer to is the *TCP connection* between *Varnish* and the backend. When a `return(pipe)` is executed, the raw bytes are shuffled over the wire, without interpreting anything as HTTP.

Here's how you detect *websockets* in *VCL*, and how you successfully *pipe* the request to the backend without the loss of the *connection upgrade headers*:

```
sub vcl_recv {
    if (req.http.upgrade ~ "(?i)websocket") {
        return (pipe);
    }
}

sub vcl_pipe {
    if (req.http.upgrade) {
        set bereq.http.upgrade = req.http.upgrade;
        set bereq.http.connection = req.http.connection;
    }
}
```

4.6.7 Enabling ESI support

Edge Side Includes are a powerful *hole-punching technique* to dissect web pages into separate blocks that are processed as individual *HTTP requests*.

The *ESI tag* is a placeholder that is interpreted by *Varnish* and is replaced by the resource it refers to.

We already talked about this, but as a reminder, this is what an *ESI tag* looks like:

```
<esi:include src="/header" />
```

Varnish can interpret these tags, but this needs to be triggered through `set beresp.do_esi = true`. Because this is more computationally intensive, you don't want to keep this turned on all the time.

Inspect the URL

In a lot of cases, people will match the URLs where *ESI parsing* is required, which might look like this:

```
vcl 4.1;

sub vcl_backend_response {
    if(bereq.url == "/" || bereq.url ~ "^/articles") {
        set beresp.do_esi = true;
    }
}
```

Unfortunately, this doesn't offer you a lot of flexibility: whenever changes in the origin application occur, the *VCL file* needs to be modified. From a *developer empowerment* point of view, this is a poor implementation.

Inspect the Content-Type header

Another approach is to make an assumption about what kind of content would require *ESI parsing*.

The example below looks at the `Content-Type`, and assumes that all *HTML* pages are *ESI parsing* candidates. So if `Content-Type: text/html` is set, *ESI parsing* is enabled:

```
vcl 4.1;

    sub vcl_backend_response {
    if (beresp.http.content-type ~ "text/html") {
        set beresp.do_esi = true;
    }
}
```

But again, this results in far too many non-ESI pages being processed.

Surrogate headers

The preferred solution takes us all the way back to *chapter 3*, where we talked about the capabilities of *HTTP*. The *surrogate headers* enable the capability that is most relevant to this use case: by leveraging the `Surrogate-Capability`, and the `Surrogate-Control` headers, you can negotiate about behavior *on the edge*.

Varnish can announce *ESI support* through the following request header:

```
Surrogate-Capability: varnish="ESI/1.0"
```

When the origin has detected *ESI support on the edge*, it can leverage this and request *ESI parsing* through the following response header:

```
Surrogate-Control: content="ESI/1.0"
```

There is in fact a *handshake* that takes place to negotiate *ESI parsing*. Here is the *VCL* required to support this:

```
vcl 4.1;

sub vcl_recv {
    set req.http.Surrogate-Capability = "varnish=ESI/1.0";
}

sub vcl_backend_response {
    if (beresp.http.Surrogate-Control ~ "ESI/1.0") {
        unset beresp.http.Surrogate-Control;
        set beresp.do_esi = true;
    }
}
```

And this is a conventional solution that only consumes CPU cycles to parse ESI when it's absolutely necessary.

4.6.8 Protocol detection

Varnish Cache doesn't support *native TLS*; *Varnish Enterprise* does. However, the most common way to support *TLS* in *Varnish* is by terminating it using a *TLS proxy*. We'll discuss this in-depth in the TLS section of *chapter 7*.

But for now, it is important to know that *Varnish* usually only processes *plain HTTP*. But thanks to the *PROXY protocol*, *Varnish* has more information about the original connection that was made.

Protocol detection and *protocol awareness* are important for the origin, because they use this information to build the right *URL schemes*. If `http://` is used as *URL* instead of `https://`, this might lead to *mixed content*, which is problematic from a browser point of view.

If you use a *TLS proxy* with *PROXY protocol support*, and connect it to *Varnish* using a listening socket that supports *PROXY*, *VCL* will use the connection metadata to populate the *endpoint variables* we discussed earlier in this chapter.

The following example uses the `std.port(server.ip)` expression to retrieve the *server port*. Because *Varnish* only does *HTTP*, this is not always `80`. If *Varnish* receives a connection via the *PROXY protocol*, the value might be `443` if a *TLS proxy* terminated the connection:

```
vcl 4.1;

import std;

sub vcl_recv {
    set req.http.X-Forwarded-Port = std.port(server.ip);

    if(req.http.X-Forwarded-Port == "443") {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

The result of this *VCL snippet* is the `X-Forwarded-Proto` header being sent to the origin. This header is a conventional one and contains either `http`, or `https`. It's up to the origin to interpret this header and act accordingly. This value can be used to force *HTTPS redirection*, but also to create the right *URLs* in *hypermedia resources*.

Using vmod_proxy

If your *TLS proxy* communicates with *Varnish* over the *PROXY protocol*, you can leverage `vmod_proxy` to easily check whether or not *TLS/SSL* was used for the request.

```
vcl 4.1;

import proxy;

sub vcl_recv {
    if(proxy.is_ssl()) {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

As you can see, it's only a matter of checking `proxy.is_ssl()`, and you're good to go.

Using vmod_tls

If you're using a recent version of *Varnish Enterprise*, *native TLS* will be supported. If you've enabled native TLS using the `-A` flag, there is no *TLS proxy*, and the *PROXY protocol* isn't used.

In *Varnish Enterprise* there is `vmod_tls` to check *TLS parameters* when native TLS is used.

Here's the `vmod_tls` equivalent of `proxy.is_ssl()`:

```
vcl 4.1;

import tls;

sub vcl_recv {
    if(tls.is_ssl()) {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

Instead of using `proxy.is_ssl()`, there's `tls.is_ssl()` to figure out what protocol was used.

4.6.9 VCL cache variations

Cache variations were discussed in *chapter 3*. Using the `Vary` header, an origin server can instruct *Varnish* to create a cache variation for a specific request header. `Vary: Accept-Language` would create a variation per cached object based on the browser language.

Although it is a very powerful instrument, a lot of web applications don't use it. If refactoring your application to include `Vary` is impossible or too hard, you can also create the variation in *VCL*.

Protocol cache variations

What better way to illustrate *VCL cache variations* than by grabbing the previous example and creating a *cache variation* on `X-Forwarded-Proto`:

```
vcl 4.1;

import std;

sub vcl_recv {
    set req.http.X-Forwarded-Port = std.port(server.ip);

    if(req.http.X-Forwarded-Port == "443") {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}

sub vcl_hash (
    hash_data(req.http.X-Forwarded-Proto);
}
```

What we're basically doing is adding `X-Forwarded-Proto` to the hash using `hash_data()`. Because we're not returning anything in `vcl_hash`, we're falling back on the *built-in VCL*, which also adds the *request URL* and the *host*.

Language cache variations

Let's grab yet another example from this chapter to illustrate *language cache variations*. Remember the example where we sanitized the `Accept-Language` header? Let's use this example to create a *cache variation*:

```
vcl 4.1;

import accept;

sub vcl_init {
    new lang = accept.rule("en");
    lang.add("nl");
}

sub vcl_recv {
    set req.http.Accept-Language = lang.filter(req.http.Accept-Lan-
guage);
}

sub vcl_hash (
    hash_data(req.http.Accept-Language);
}
```

Because `Accept-Language` is sanitized, the number of values are limited, which reduces the number of *cache variations*. You can confidently *vary* on this header. And if you don't, you can still use `hash_data(req.http.Accept-Language)` to do it in *VCL*.

4.6.10 Language cookie cache variation

However, the majority of multilingual websites use a language selection menu, or a splash page, instead of the `Accept-Language` header. The selected language is then stored in a *cookie*.

But we know that *Varnish* doesn't cache when cookies are present because it implies stateful content. Varying on the `Cookie` header is also a bad idea, given the amount of tracking cookies that are injected.

But all is not lost! We can extract the value of the *language cookie*, and create a variation using *VCL*.

Imagine this being the language cookie:

```
Cookie: language=en
```

This is the *VCL code* you could use to *vary* on the language value:

```
vcl 4.1;

sub vcl_recv {
    if (req.http.Cookie) {
        set req.http.Cookie = ";" + req.http.Cookie;
        set req.http.Cookie = regsuball(req.http.Cookie, "; +", ";");
        set req.http.Cookie = regsuball(req.http.Cookie, ";(lan-
guage)=", "; \1=");
        set req.http.Cookie = regsuball(req.http.Cookie, ";[^ ]
[^;]*", "");
        set req.http.Cookie = regsuball(req.http.Cookie, "^[; ]+|[;
]+$", "");

        if (req.http.cookie ~ "^\s*$") {
            unset req.http.cookie;
        }

        return(hash);
    }
}

sub vcl_hash {
    if(req.http.Cookie ~ "^.*language=(nl|en|fr);*.*$") {
        hash_data(regsub( req.http.Cookie, "^.*language=(nl|en|-
fr);*.*$", "\1" ));
    } else {
        hash_data("en");
    }

}
```

In the `vcl_recv` subroutine, we're doing the typical find and replace magic where we delete all the cookies, except the ones that matter. In our case that's the *language cookie*.

Instead of doing a `return(pass)` when there are still cookies left, we deliberately call `return(hash)`, and consider the content cacheable.

In `vcl_hash`, we check whether or not the cookies have been set. If not, we add `en` as the default *language cache variation*. Otherwise, we just extract the value from the cookie using the `regsub()` function.

Because we explicitly defined the list of supported languages in the regular expression, we avoid that too many variations can occur.

Using vmod_cookie

Here's the same example, but with `vmod_cookie` for those who are on *Varnish Cache 6.4 or later*:

```
vcl 4.1;

import cookie;

sub vcl_recv {
    cookie.parse(req.http.cookie);
    cookie.keep("language");
    set req.http.cookie = cookie.get_string();
    if (req.http.cookie ~ "^\s*$") {
         unset req.http.cookie;
     }
}

sub vcl_hash {
    if(cookie.get("language") ~ "^(nl|en|fr|de|es)$" ) {
        hash_data(cookie.get("language"));
    } else (
        hash_data("en");
    }

}
```

Using vmod_cookieplus

Here's the `vmod_cookieplus` implementation for those who use *Varnish Enterprise*:

```
vcl 4.1;

import cookieplus;

sub vcl_recv {
    cookieplus.keep("language");
    cookieplus.write();
}

sub vcl_hash {
    if(cookieplus.get("language") ~ "^(nl|en|fr|de|es)$" ) {
        hash_data(cookieplus.get("language"));
    } else (
        hash_data("en");
    }

}
```

4.6.11 Custom error messages

When a backend response fails, *Varnish* will return an error page that looks like this:

```
Error 503 Backend fetch failed
Backend fetch failed

Guru Meditation:
XID: 3

---

Varnish cache server
```

It looks a bit weird, and the *guru meditation* message doesn't look that appealing.

The current built-in VCL implementation

These error messages, and the layout for *synthetic responses* are part of the *built-in VCL*. Here's the *VCL code* for `vcl_backend_error`, in case of errors:

```
sub vcl_backend_error {
    set beresp.http.Content-Type = "text/html; charset=utf-8";
    set beresp.http.Retry-After = "5";
    set beresp.body = {"<!DOCTYPE html>
<html>
  <head>
    <title>"} + beresp.status + " " + beresp.reason + {"</title>
  </head>
  <body>
    <h1>Error "} + beresp.status + " " + beresp.reason + {"</h1>
    <p>"} + beresp.reason + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + bereq.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"};
    return (deliver);
}
```

Regular *synthetic responses* triggered from *client-side VCL logic* have a similar *VCL implementation*:

```
sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";
    set resp.body = {"<!DOCTYPE html>
<html>
  <head>
    <title>"} + resp.status + " " + resp.reason + {"</title>
  </head>
  <body>
    <h1>Error "} + resp.status + " " + resp.reason + {"</h1>
    <p>"} + resp.reason + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + req.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"};
    return (deliver);
}
```

Customize error messages using templates

To tackle the issue, you could modify the string that is assigned to `beresp.body` in `vcl_backend_response`, or `resp.body` in `vcl_synth`, but that can go wrong really quickly.

Not only can it become a *copy-paste mess*, but you also have to take *variable interpolations* into account.

The ideal solution is to load a template from a file, potentially replace some placeholder values, and inject the string value into the response body.

Here's the *VCL code*:

```
vcl 4.1;

import std;

sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";
    set resp.body = regsuball(std.fileread("/etc/varnish/synth.htm-
l"),"<<REASON>>",resp.reason);
    return (deliver);
}

sub vcl_backend_error {
    set beresp.http.Content-Type = "text/html; charset=utf-8";
    set beresp.http.Retry-After = "5";
    set beresp.body = regsuball(std.fileread("/etc/varnish/synth.htm-
l"),"<<REASON>>",beresp.reason);
    return (deliver);
}
```

This example will use `std.fileread()` to load a file from disk and present it as a string. Using `regsuball()` we're going to replace all occurrences of the `<<REASON>>` placeholder in that file with the actual *reason phrase*. This will be provided by either `resp.reason` or `beresp.reason`.

The cool thing about this implementation is that you can have your frontend developers compose and style this file to match the branding of the actual website. It can contain images, CSS, JavaScript, and all the other goodies, but it doesn't fill up your *VCL* with very verbose content.

4.6.12 Caching objects on the second miss

Directly inserting an object in cache when a *cache miss* occurs is the default *Varnish* behavior. It also makes sense: we want to avoid hitting the origin server, so caching something as soon as possible is the logical course of action.

However, when you have a limited cache size, inserting random objects may not be very efficient. For long-tail content, you risk filling up your cache with objects that will hardly be requested.

A solution could be to only *insert an object in cache on the second miss*. The following example leverages `vmod_utils`, which is a *Varnish Enterprise VMOD*.

```
vcl 4.1;

import utils;

sub vcl_backend_response {
    if (!utils.waitinglist() && utils.backend_misses() == 0) {
        set beresp.uncacheable = true;
        set beresp.ttl = 24h;
    }
}
```

When a *cache miss* occurs, and we fetch content from the origin, `utils.backend_misses()` will tell us whether or not a *hit-for-miss* has already occurred.

As long as this value is `0`, we know that this resource has not been requested, and didn't result in a *hit-for-miss* for the last 24 hours. In that case we will enable `beresp.uncacheable` and set the *TTL* to *24 hours*.

This ensures that *Varnish* keeps track of that *hit-for-miss* for a full day. When the next request for that resource is received during that timeframe, we know it's somewhat popular, and we can insert the response in cache.

Because of *request coalescing*, it is possible that other clients are requesting the same content at exactly the same time. These requests will be put on the *waiting list* while the first in line fetches the content. We cannot *hit-for-miss* when this happens, because that would cause *request serialization*. Luckily `utils.waitinglist()` gives us insight into the number of *waiting requests* for that resource.

The end result is that only *hot content* is cached, and our precious caching space is less likely to be wasted on long-tail content. Of course you can tune this behavior and choose to only cache on the third or fourth miss. That's up to you.

Keep in mind that this doesn't work with *hit-for-pass*: when you add `return(pass)` to this logic to trigger *hit-for-pass*, the decision not to cache will be remembered for *24 hours* and cannot be undone by the next cacheable response. This defeats the purpose of this feature.

4.7 Validation and testing

Working with *Varnish* and writing *VCL* is just like any other development project: there's code involved, and it will be deployed to a runtime that can be configured.

There are a lot of moving parts, so a *quality assurance* process should be in place.

This *QA process* ensures that the expectations are met, and that the deployed *VCL* results in the desired behavior.

There are a couple of ways to approach this. In this section we'll discuss *syntax validation* and *testing*.

4.7.1 Syntax validation

If you want check the syntactical correctness of your *VCL code*, you can use the `varnishadm vcl.load` command. This command takes two arguments: the name of the configuration and the VCL file:

```
~# varnishadm vcl.load myconfig default.vcl
VCL compiled.
```

When *Varnish* fails to compile your *VCL file*, the `varnishadm` command will show you an error message:

```
~# varnishadm vcl.load myconfig default.vcl
Message from VCC-compiler:
Expected ‘;’ got ‘}’
(program line 381), at
(‘/etc/varnish/default.vcl’ Line 22 Pos 1)
}
#

Running VCC-compiler failed, exited with 2
VCL compilation failed
Command failed with error code 106
```

We casually use `myconfig` as the configuration name. Not only do we advise you to figure out a more appropriate name, we want to warn you that you'll get the following error message when you try to load the same configuration name twice:

```
~# varnishadm vcl.load myconfig default.vcl
Already a VCL named myconfig
Command failed with error code 106
```

After you've done your syntax checks using the `vcl.load` subcommand, we advise you to *discard* the load configurations. Not only do you risk name clashes, but loaded configurations also consume resources.

Here's how you discard the `myconfig` *VCL configuration*:

```
varnishadm vcl.discard myconfig
```

You can also use `varnishadm vcl.inline` and pass a string containing your *VCL* code. Here's an example within the `varnishadm` shell:

```
varnish> vcl.inline myconfig2 << EOF
> vcl 4.1;
>
> backend default {
>     .host = "localhost";
>     .port = "8080";
>}
> EOF
200
VCL compiled.
```

You can also do the same from outside of the `varnishadm` shell, but that requires two levels of *HEREDOC*:

> In Bash, a Here document allows multiple lines to be passed as input to a command.

```
~# varnishadm << EOF
> vcl.inline myconfig2 << EOF2
> vcl 4.1;
> backend default {
>.    .host = "localhost";
>     .port = "8080";
> }
> EOF2
> EOF
200
VCL compiled.
```

4.7.2 Testing

A syntax check is good, but you can hardly consider it your only *QA* task. Logical errors aren't detected. Testing the behavior of your *VCL code* is just as important.

The `varnishtest` program is a script-driven testing tool that uses *Varnish Test Case (VTC)* files to test specific scenarios. It is a *functional testing* tool, where the various elements of the delivery chain can be defined:

- The *Varnish* server
- The client
- The origin server
- Extra proxy servers
- Varnishlog output
- Other server processes that are called

Built-in VCL test

The first example of a *VTC* file is the following example:

```
varnishtest "Check that a cache fetch + hit transaction works"

server s1 {
    rxreq
    txresp -hdr "Connection: close" -body "012345\n"
} -start

varnish v1 -vcl+backend { } -start

client c1 {
    txreq -url "/"
    rxresp
    expect resp.status == 200
} -run

client c2 {
    txreq -url "/"
    rxresp
    expect resp.status == 200
} -run

# Give varnish a chance to update stats
delay .1

varnish v1 -expect sess_conn == 2
varnish v1 -expect cache_hit == 1
varnish v1 -expect cache_miss == 1
varnish v1 -expect client_req == 2
```

This *VTC* file is executed using the following command:

```
~# varnishtest test.vtc
#    top  TEST test.vtc passed (5.446)
```

And as you can see, the test passed.

But what are we actually testing?

- A server is defined that closes the connection after the request and returns the `012345\n` response body.
- A Varnish server is defined that connects with the backend.
- The first client connects to the homepage and expects an *HTTP 200* response.
- The second client also connects to the homepage and expects an *HTTP 200* response.
- In the end the following expectations have to be met:
  - The total number of connections is two.
  - There was only one cache hit that occurred.
  - There was only one cache miss that occurred.
  - Two clients connect to Varnish in total.

If these expectations are met, the test passes.

A failing test

Imagine the following *VTC* where we deliberately cause a failure:

```
varnishtest "A failing test"
server s1 {
    rxreq
    txresp -status 400
} -start

varnish v1 -vcl+backend { } -start

client c1 {
    txreq
    rxresp
    expect resp.status == 200
} -run
```

The expectation is that the status code will be an *HTTP 200*. However, the server returns an *HTTP 400* status code.

This is the command's output:

```
~# /etc/varnish# varnishtest -q failed.vtc
#    top  TEST failed.vtc FAILED (5.245) exit=2
```

Looking at Varnish's tests

VTC files aren't only there for your convenience and to test your implementation. The testing framework is also used by *Varnish* core developers for both the open source and the enterprise version of the software.

The *master* branch of *Varnish Cache* has about 850 *VTC files* in its *GitHub repository*. For *Varnish Enterprise* the number is around 1460.

The *GitHub repository* for *Varnish Enterprise* is not open source, of course, but you can definitely have a look at https://github.com/varnishcache/varnish-cache/tree/master/bin/varnishtest/tests to get inspired.

Here's the content from the `README` file in that repository, which explains the naming scheme that was used. If you understand the naming scheme, you'll be able to find the kind of test you're looking for.

```
Naming scheme
-------------

    The intent is to be able to run all scripts in lexicographic
    order and get a sensible failure mode.

    This requires more basic tests to be earlier and more complex
    tests to be later in the test sequence, we do this with the
    prefix/id letter:

        [id]%05d.vtc

    id ~ ^a     --> varnishtest(1) tests
    id ~ ^a02   --> HTTP2
    id ~ ^b     --> Basic functionality tests
    id ~ ^c     --> Complex functionality tests
    id ~ ^d     --> Director VMOD tests
    id ~ ^e     --> ESI tests
    id ~ ^f     --> Security-related tests
    id ~ ^g     --> GZIP tests
    id ~ ^h     --> HAproxy tests
    id ~ ^i     --> Interoperability and standards compliance
    id ~ ^j     --> JAIL tests
    id ~ ^l     --> VSL tests
    id ~ ^m     --> VMOD tests excluding director
    id ~ ^o     --> prOxy protocol
    id ~ ^p     --> Persistent tests
    id ~ ^r     --> Regression tests, same number as ticket
    id ~ ^s     --> Slow tests, expiry, grace, etc.
    id ~ ^t     --> Transport protocol tests
    id ~ ^t02   --> HTTP2
    id ~ ^u     --> Utilities and background processes
    id ~ ^v     --> VCL tests: execute VRT functions
```

A VCL test

Remember that *language cookie variation* example in the *Making changes* section of this chapter?

To be sure the variations are respected by the *VCL*, we could run the following test:

```
varnishtest "Language cookie cache variation"
server s1 {
    rxreq
    expect req.http.Cookie == "language=nl"
    txresp -body "Goede morgen"
    rxreq
    expect req.http.Cookie != "language=nl"
    txresp -body "Good morning"

    rxreq
    expect req.http.Cookie != "language=nl"
    txresp -body "Good morning"

    rxreq
    expect req.http.Cookie != "language=nl"
    txresp -body "Good morning"
} -start

varnish v1 -vcl+backend {
    vcl 4.1;

    sub vcl_recv {
        if (req.http.Cookie) {
            set req.http.Cookie = ";" + req.http.Cookie;
            set req.http.Cookie = regsuball(req.http.Cookie, "; +",
";");
            set req.http.Cookie = regsuball(req.http.Cookie, ";(lan-
guage)=", "; \1=");
            set req.http.Cookie = regsuball(req.http.Cookie, ";[^ ]
[^;]*", "");
            set req.http.Cookie = regsuball(req.http.Cookie, "^[;
]+|[; ]+$", "");

            if (req.http.cookie ~ "^\s*$") {
                unset req.http.cookie;
            }

            return(hash);
        }
    }

    sub vcl_hash {
        if(req.http.Cookie ~ "^.*language=(nl|en|fr);*.*$") {
            hash_data(regsub( req.http.Cookie, "^.*language=(nl|en|-
fr);*.*$", "\1" ));
        } else {
            hash_data("en");
        }

    }
} -start

client c1 {
    txreq -hdr "Cookie: a=1; b=2; language=nl; c=3"
    rxresp
    expect resp.body == "Goede morgen"

    txreq -hdr "Cookie: a=1; language=en"
    rxresp
    expect resp.body == "Good morning"

    txreq -hdr "Cookie: a=1; language=fr"
    rxresp
    expect resp.body == "Good morning"

    txreq
    rxresp
    expect resp.body == "Good morning"
} -run
```

This example has a *server* that will return `Goede morgen` in Dutch when the `language` cookie equals `nl`. In all other cases, we will fall back to English and return `Good morning`.

The *client* will send requests containing various cookies: both irrelevant cookies, and the relevant `language` cookie. Supported values will be sent, like `nl` and `en`. We'll also send an unsupported value, like `fr`. And we'll even test what happens if no `Cookie` header is sent.

This test passes, so the *VCL* matches our expectations and correctly supports *language cookie cache variations*.

4.8 Summary

Congratulations! You're at the end of *chapter 4*, in which we covered the ins and outs of *VCL*.

At this point, you should be comfortable with *VCL*. You should be able to understand its syntax, you should know about the subroutines, the return statements, and the variables.

The main objective is to understand how *Varnish* leverages the *finite state machine*, how the *built-in VCL* is used to control states and transitions, and how you can extend that behavior with *VCL*.

Although there's educational value to this chapter, you cannot consider it as documentation. If you need documentation for *VCL*, you'll find it at http://varnish-cache.org/docs/6.0/reference/vcl.html#varnish-configuration-language.

Every chapter from now on will use *VCL*, so be ready to apply what you learned.

Let's get ready for *chapter 5* where we'll explain what *VMODs* are, which ones are shipped by default, and how you can use them. We'll even show you how to write your own *VMODs*.

Chapter 5: Varnish Modules (VMODs)

The *Varnish Configuration Language* is a very powerful programming language, but the scope of its features is entirely focused on the *finite state machine*, on *HTTP requests*, *HTTP responses*, *caching objects*, and *backend fetches*.

It is a *domain-specific language*, and due to its scope, you might run into limitations.

For example, you cannot look up a *client IP address* in a database for access control purposes. It is also not possible to load balance using a pool of backends using pure *VCL*.

Luckily *Varnish* has a good solution for these types of limitations, and the solution comes in the form of *Varnish Modules*, or *VMODs* as we tend to call them.

VMODs are the focus of this chapter.

5.1 What's a VMOD?

A *VMOD* is a *shared library* that is written in `C`. It has a set of functions, containing the logic it wants to expose. These *functions* can then be imported and called from *VCL*, which in turn adds new functionality to *VCL*.

This is a very powerful concept because anything that can be written in `C` can in fact be exposed to *VCL*.

Varnish isn't just a cache: by using the right *VMODs*, you can reshape content, route traffic, use custom authentication mechanisms and implement all kinds of custom logic using powerful *VMODs*.

5.1.1 Scope and purpose

However, it is important to know that *VMODs* aren't a gateway to *Varnish's* inner workings: the *APIs* that *Varnish* provides are quite limited, and there aren't a lot of hooks. *VMODs* are intended to act independently, and usually wrap around some sort of library or logic in a simple and easy-to-use manner.

Some of the *VMODs* that are shipped with *Varnish* go beyond basic *wrapping* and do interface with the *Varnish core*. But that's because the core has patches to open up access to internal APIs, purpose-built for these *VMODs*

The majority of *VMODs* perform the following tasks:

- String manipulation
- Type casting
- Extracting values from complex data types
- Inspecting session and request information
- Safe access to third-party libraries

5.1.2 VMOD API

Every *VMOD* has an *API*. It is a collection of function calls and objects that the module exposes to *VCL*.

The *VCL API* is kept in a `.vcc` file, and one or more `.c` files contain the actual code.

Take for example https://github.com/varnishcache/libvmod-example, where you find the source code of the `vmod_example` *VMOD*.

`vmod_example` is a sample module for aspiring *VMOD* writers who need a bit of inspiration. The module itself doesn't really do much, but it does lead the way, contains the necessary files, and scripts to build the module.

In the `src` directory, you'll find a `vmod_example.vcc` file that contains the API.

```
$Module example 3 Example VMOD

DESCRIPTION
===========

This is the embedded documentation for the example VMOD. It should
explain
the purpose and what problems it solves, with relevant examples.

It can span multiple lines and is written in RST format.
You can even have links and lists in here:

* https://github.com/varnish/libvmod-example/
* https://www.varnish-cache.org/

$Event event_function
$Function STRING info()

Returns a string set by the last VCL event, demonstrating the use of
event functions.

$Function STRING hello(STRING)

The different functions provided by the VMOD should also have their
own
embedded documentation. This section is for the hello() function.
```

The `$Module` line defines the name of the *VMOD*, and the `$Function` lines define the *API*. The example above contains two functions:

- `example.info()` which returns the information about the last *VCL event* that occurred
- `example.hello()` which performs the typical *Hello World*, based on an input argument

The corresponding code can be found in `vmod_example.c`. The *C-code* itself isn't that important to *VMOD* users, because the idea is that the *VCL functions* are the interface, and the implementation is abstracted by these *VCL functions*. As long as the documentation for the *VMOD* is good, the code is irrelevant.

5.1.3 VCL usage

Our example *VMOD* can be loaded into our *VCL file* by using the `import` statement:

```
vcl 4.1;

import example;

sub vcl_deliver {
    set resp.http.hello = example.hello("Thijs");
}
```

As you can see the `example.hello()` function becomes available and can be used in any of the *VCL subroutines*. In the example above, we're using it to set the following response header:

```
hello: Hello Thijs
```

5.1.4 VMOD initialization

VMODs aren't always a collection of utility functions. Often they keep track of state and are grouped into one or more objects.

Setting them up may require an initialization stage, which is done in the `vcl_init` subroutine. Throughout the book, we haven't mentioned this subroutine a lot.

Let's immediately throw in an example where `vcl_init` initializes `vmod_directors`:

```
vcl 4.1;

import directors;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

sub vcl_init {
    new vdir = directors.round_robin();
    vdir.add_backend(backend1);
    vdir.add_backend(backend2);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend();
}
```

This *VMOD* creates a director object that groups multiple backends into one. The *director* will distribute load across these backends using a distribution algorithm, and will expose itself as a single backend using the `.backend()` function.

Adding backends and choosing the right distribution algorithm is all done in `vcl_init`. As you can see in the example above, we're using the `directors.round_robin()` function to create a directors object named `vdir`. This object uses the `vdir.add_backend()` method to assign backends.

> `vmod_directors` is a load-balancing *VMOD* and will be covered in detail in *chapter 7*.

5.1.5 Installing a VMOD

Installing a *VMOD* is a lot like compiling *C-code*. That's because a *VMOD* is written in `C`.

VMODs come with an `autogen.sh` script that will inspect your operating system, and will determine where to find the `libtoolize`, `autoconf`, and `automake` tools. The script also looks for the `varnishapi` library.

After this script finishes its execution, you can run the `configure` script that was generated. This script will configure the *GCC compiler*.

Eventually, all the configurations are in place to run the `make` command, which will use a `Makefile` to compile specific source files.

The final step is running `make install`, which will turn the compiled files into `libvmod_example.so`, and put this *shared object* in the right directory. By default this is `/usr/lib/varnish/vmods/`.

In order to successfully compile and install a *VMOD* your build system will have some dependencies.

If you're on a *Debian* or *Ubuntu* system, you can use the following command to install the required dependencies:

```
# apt update && apt install -y \
varnish build-essential automake libtool python3-docutils
```

If you're on *Red Hat*, *Fedora*, or *CentOS*, this is the equivalent:

```
# yum check-update && yum install -y \
varnish-devel gcc make automake libtool python3-docutils
```

It's worth mentioning that you generally don't have to compile or install any *VMODs* yourself, as most of them are packaged either with *Varnish Cache* or with *Varnish Enterprise*. It's only for when you are developing your own *VMODs*, or when you are using *non-packaged community VMODs*.

5.2 Which VMODs are shipped with Varnish Cache?

VMODs can be built and installed separately, but *Varnish* also ships a couple of *VMODs* on its own.

Varnish Cache has a set of *in-tree VMODs* that are part of the source code. This means that these *VMODs* are included in the standard installation.

It's quite easy to spot them. When you go to https://github.com/varnishcache/varnish-cache, you'll see them in the `vmod` folder:

VMOD name	Description
`vmod_blob`	Utilities for encoding and decoding *BLOB* data in VCL
`vmod_cookie`	Inspect, modify, and delete client-side cookies
`vmod_directors`	Directors group multiple backends as one, and load balance backend requests between these backends using a variety of load-balancing algorithms
`vmod_proxy`	Retrieve TLS information from connections made using the *PROXY protocol*
`vmod_purge`	Perform *hard* and *soft* purges
`vmod_std`	A library of basic utility functions to perform conversions, interact with files, perform custom logging, etc.
`vmod_unix`	Get the *user*, the *group*, the *gid*, and the *uid* from connections made over *Unix domain sockets (UDS)*
`vmod_vtc`	A utility module for `varnishtest`

This list comes from the `master` branch of this *Git repository*. It represents the current state of the open source project. As mentioned, *Varnish Software* maintains a *6.0 LTS version* of *Varnish Cache*. In this version all *VMODs* from the open source project are included, except `vmod_cookie`, which has been replaced by `vmod_cookieplus`.

5.2.1 vmod_blob

A *BLOB* is short for a *Binary Large Object*. It's a data type that is used for the *hash keys* and for the *response body*.

Here's an example where we transfer `req.hash`, a *BLOB* that represents the *hash key* of the request, into a *string* value:

```
vcl 4.1;

import blob;

sub vcl_deliver {
    set resp.http.x-hash = blob.encode(encoding=BASE64,blob=req.
hash);
}
```

The `blob.encode()` function is used for the conversion. The name of the function indicates that an encoding format is required. We use `base64`, which is a common encoding format suitable for use in HTTP header fields.

`blob.encode()` has three arguments, but we only set two. The second argument has been omitted. It is the `case` argument that defaults to `DEFAULT`. But because we're using *named arguments*, it is perfectly fine to omit arguments.

When we request `http://localhost/`, the corresponding `x-hash` header is the following:

```
x-hash: 3k0f0yRKtKt7akzkyNsTGSDOJAZOQowTwKWhu5+kIu0=
```

`vmod_blob` has plenty of other functions and methods. In the example above we used `blob.encode()`; there's also a `blob.decode()`, which converts a *string* into a *blob*. All these functions and methods can be found at https://varnish-cache.org/docs/6.0/reference/vmod_generated.html#vmod-blob.

5.2.2 vmod_cookie

`vmod_cookie` is only available as of *Varnish Cache 6.4*, and it facilitates interaction with the `Cookie` header. You've already seen a couple of examples where this *VMOD* was used to remove and to get cookies.

Imagine the following `Cookie` header:

```
Cookie: language=en; accept_cookie_policy=true;
_ga=GA1.2.1915485056.1587105100; _gid=GA1.2.71561942.1601365566; _
gat=1
```

Here's what we want to do:

- If `accept_cookie_policy` is not set, redirect to the homepage
- If `/fr` is called, set the `language` cookie to `fr`
- Remove the tracking cookies

Here's the *VCL* code to achieve this:

```
vcl 4.1;

import cookie;

sub vcl_recv {
    cookie.parse(req.http.Cookie);
    if(!cookie.isset("accept_cookie_policy")) {
        return(synth(301,"/"));
    }
    if (req.url ~ "^/fr/?" && cookie.get("language") != "fr") {
        cookie.set("language","fr");
    }
    cookie.filter_re("^_g[a-z]{1,2}$");
    set req.http.Cookie = cookie.get_string();
}

sub vcl_synth {
    if (resp.status == 301) {
        set resp.http.location = resp.reason;
        set resp.reason = "Moved";
        return (deliver);
    }
}
```

The `return(synth(301,"/"))` in conjunction with the `vcl_synth` logic allows you to create a custom *HTTP 301* redirect.

The rest of the API and more `vmod_cookie` examples can be found here: http://varnish-cache.org/docs/trunk/reference/vmod_cookie.html

5.2.3 vmod_directors

`vmod_directors` is a load-balancing *VMOD*. It groups multiple backends and uses a distribution algorithm to balance requests to the backends it contains.

We'll briefly cover two load balancing examples using this *VMOD*, but in *chapter 7* there will be a dedicated section about load balancing.

You already know the next example because we covered it in the *VMOD initialization* section:

```
vcl 4.1;

import directors;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

sub vcl_init {
    new vdir = directors.round_robin();
    vdir.add_backend(backend1);
    vdir.add_backend(backend2);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend();
}
```

We initialize the *director* in `vcl_init`, where we choose the *round-robin* distribution algorithm to balance load across `backend1` and `backend2`.

For the second example, we're going to take the same *VCL*, but instead of a *round-robin* distribution, we're going for a *random* distribution. The changes aren't that big though:

```
vcl 4.1;

import directors;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

sub vcl_init {
    new vdir = directors.random();
    vdir.add_backend(backend1,10);
    vdir.add_backend(backend2,20);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend();
}
```

The example above uses a *random* distribution of the load, but not with equal weighting:

- `backend1` will receive *33%* percent of all the requests
- `backend2` will receive *66%* percent of all the requests

And that's because of the *weight* arguments that were added to each backend of the director. The equation for *random* is as follows: `100 * (weight / (sum(all_added_weights)))`.

The rest of the API, and more director examples can be found here: https://varnish-cache.org/docs/6.0/reference/vmod_generated.html#vmod-directors

5.2.4 vmod_proxy

`vmod_proxy` is used to extract *client-* and *TLS-* information from a request to *Varnish* via the *PROXY protocol*.

Imagine the following *Varnish runtime parameters*:

```
varnishd -a:80 -a:8443,PROXY -f /etc/varnish/default.vcl
```

Here's what this means:

- Varnish accepts regular *HTTP* connections on port *80*.
- Varnish also accepts connections on port *8443*, which are made using the *PROXY protocol*.
- The *VCL file* is located at `/etc/varnish/default.vcl`.

Assuming the *PROXY connection* was initiated by a *TLS proxy*, we can use `vmod_proxy` to extract *TLS information* that is transported by the *PROXY protocol*.

Here's a *VCL* example that extracts some of the information into custom response headers:

```
vcl 4.1;
import proxy;

sub vcl_deliver {
    set resp.http.alpn = proxy.alpn();
    set resp.http.authority = proxy.authority();
    set resp.http.ssl = proxy.is_ssl();
    set resp.http.ssl-version = proxy.ssl_version();
    set resp.http.ssl-cipher = proxy.ssl_cipher();
}
```

And this is some example output, containing the custom headers:

```
alpn: h2
authority: example.com
ssl: true
ssl-version: TLSv1.3
ssl-cipher: TLS_AES_256_GCM_SHA384
```

This is what we learn about the *SSL/TLS connection* through these values:

- A successful *TLS/SSL* connection was made.
- The communication protocol is `HTTP/2`.
- *SNI* determined that `example.com` is the authoritative hostname.
- The *SSL/TLS* version we're using is `TLSV1.3`.
- The *SSL/TLS* cipher is an *Advanced Encryption Standard with 256bit key in Galois/Counter mode*. The hashing was done using a *384-bit Secure Hash Algorithm*.

5.2.5 vmod_std

`vmod_std` is the *standard VMOD* that holds a collection of *utility functions* that are commonly used in everyday scenarios. Although these could have been *native VCL function*, they were put in a *VMOD* nevertheless.

`vmod_std` performs a variety of tasks, but its functions can be grouped as follows:

- String manipulation
- Type conversions
- Logging
- File access
- Environment variables
- Data extraction from complex types

Only a couple of examples for this *VMOD* have been added, but the full list of functions can be consulted here: https://varnish-cache.org/docs/6.0/reference/vmod_generated.html#varnish-standard-module.

Logging

Let's start with an example that focuses on logging, but that uses other functions as utilities:

```
vcl 4.1;

import std;

sub vcl_recv {
    if (std.port(server.ip) == 443) {
        std.log("Client connected over TLS/SSL: " + server.ip);
        std.syslog(6,"Client connected over TLS/SSL: " + server.ip);
        std.timestamp("After std.syslog");
    }
}
```

- `std.log()` will add an item to the *Varnish Shared Memory Log (VSL)* and tag it with a `VCL_Log` tag.
- `std.timestamp()` will also add an item to the *VSL*, but will use a `Timestamp` tag, and will drop in a timestamp for measurement purposes.
- `std.syslog()` will add a log item to the *syslog*.

Here's the *VSL* output that was captured using `varnishlog`. You clearly see the `std.log()` and `std.timestamp()` string values in there:

```
-   VCL_Log        Client connected over TLS/SSL: 127.0.0.1
-   Timestamp      After std.syslog: 1601382665.510435 0.000147
0.000140
```

When we look at the *syslog*, you'll see the log line that was triggered by `std.syslog()`

```
Sep 29 14:47:05 server varnishd[1260]: Varnish client.ip: 127.0.0.1
```

String manipulation

Let's immediately throw in an example where we combine a few string manipulation functions:

```
vcl 4.1;

import std;

sub vcl_recv {
    set req.url = std.querysort(req.url);
    set req.url = std.tolower(req.url);
    set req.http.User-Agent = std.toupper(req.http.User-Agent);
}
```

So imagine sending the following request to *Varnish*:

```
HEAD /?B=2&A=1 HTTP/1.1
Host: localhost
User-Agent: curl/7.64.0
```

Here's what's happening behind the scenes, based on a specific `varnishlog` command:

```
$ varnishlog -C -g request -i requrl -I reqheader:user-agent
*   << Request  >> 23
-   ReqURL         /?B=2&A=1
-   ReqHeader      User-Agent: curl/7.64.0
-   ReqURL         /?A=1&B=2
-   ReqURL         /?a=1&b=2
-   ReqHeader      user-agent: CURL/7.64.0
```

- The input for the *URL* is `/?B=2&A=1`
- The input for the `User-Agent` header is `curl/7.64.0`
- The *URL*'s query string arguments are sorted alphabetically, which results in `/?A=1&B=2`
- The *URL* is in lowercase, which results in `/?a=1&b=2`
- The `User-Agent` header is in uppercase, which results in `CURL/7.64.0`

Environment variables

The `std.getenv()` function can retrieve the values of *environment variables*.

The following example features an *environment variable* named `VARNISH_DEBUG_MODE`. If it is set to `1`, debug mode is enabled, and a custom `X-Varnish-Debug` header is set:

```
vcl 4.1;

import std;

sub vcl_deliver {
    if(std.getenv("VARNISH_DEBUG_MODE") == "1") {
        if(obj.hits > 0) {
            set resp.http.X-Varnish-Debug = "HIT";
        } else {
            set resp.http.X-Varnish-Debug = "MISS";
        }
    }
}
```

You can set environment variables for your systemd service with `systemd edit varnish`, and then add `Environment="MYVAR=myvalue"` under the `[Service]` section.

Reading a file

`vmod_std` has a function called `std.fileread()`, which will read a file from disk and return the string value.

We're not going to be too original with the *VCL* example. In one of the previous sections, we talked about setting a custom HTML template for `vcl_synth`. Let's take that example again:

```
vcl 4.1;

import std;

sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";
    set resp.body = regsuball(std.fileread("/etc/varnish/synth.
html"),
    "<<REASON>>",resp.reason);
    return (deliver);
}
```

Whenever `return(synth())` is called, the contents from `/etc/varnish/synth.html` are used as a template, and the `<<REASON>>` placeholder is replaced with the actual *reason phrase* that was set in `synth()`.

You could also make this conditional by using `std.file_exists()`:

```
vcl 4.1;

import std;

sub vcl_synth {
    if(std.file_exists("/etc/varnish/synth.html")) {
        set resp.http.Content-Type = "text/html; charset=utf-8";
        set resp.http.Retry-After = "5";
        set resp.body = regsuball(std.fileread("/etc/varnish/synth.
html"),
        "<<REASON>>",resp.reason);
        return (deliver);
    }
}
```

Server ports

The `IP` type in *VCL* that is returned by variables like `client.ip` doesn't just contain the string version of the IP address. It also contains the port that was used.

But when `IP` output is cast into a string, the port information is not returned. The `std.port()` function extracts the *port* from the *IP* and returns it as an *integer*.

Here's an example:

```
vcl 4.1;

import std;

sub vcl_recv {
    if(std.port(server.ip) != 443) {
        set req.http.Location = "https://" + req.http.host + req.url;
        return(synth(301,"Moved"));
    }
}

sub vcl_synth {
    if (resp.status == 301) {
        set resp.http.Location = req.http.Location;
        return (deliver);
    }
}
```

This example will check if the port that was used to connect to *Varnish* was *443* or not. Port 443 is the port that is used for *HTTPS* traffic. If this port is not used, redirect the page to the *HTTPS* equivalent.

5.2.6 vmod_unix

If a connection to Varnish is made over *UNIX domain sockets*, `vmod_unix` can be used to figure out the following details about the *UDS* connection:

- The *username* of the peer process owner
- The *group name* of the peer process owner
- The *user id* of the peer process owner
- The *group id* of the peer process owner

In the list, we refer to the *peer process owner*: this is the user that executes the process that represents the *client-side* of the communication. Because connections over *UDS* are done locally, the *client side* isn't represented by an actual client, but another proxy.

A good example of this is *Hitch*: *Hitch* is a *TLS PROXY* that is put in front of *Varnish* to terminate the *TLS* connection. For performance reasons, we can make *Hitch* connect to *Varnish* over *UDS*.

Because the *peer process* doesn't use *TCP/IP* to communicate with *Varnish*, we cannot restrict access based on the *client IP address*. However, file system permissions can be used to restrict access.

Here's how `vmod_unix` can be used to restrict access to *Varnish*:

```
vcl 4.1;

import unix;

sub vcl_recv {
    # Return "403 Forbidden" if the connected peer is
    # not running as the user "trusteduser".
    if (unix.user() != "trusteduser") {
        return(synth(403) );
    }

    # Require the connected peer to run in the group
    # "trustedgroup".
    if (unix.group() != "trustedgroup") {
        return(synth(403) );
    }

    # Require the connected peer to run under a specific numeric
    # user id.
    if (unix.uid() != 4711) {
        return(synth(403) );
    }

    # Require the connected peer to run under a numeric group id.
    if (unix.gid() != 815) {
        return(synth(403) );
    }
}
```

The `unix.user()` is used to retrieve the *username* of the user that is running the peer process. The example above restricts access if the username is not `trusteduser`.

You can also use the `unix.uid()` function to achieve the same goal, based on the *user id*, instead of the *username*. In the example above, we restrict access to *Varnish* if the *user id* is not `4711`.

And for groups, the workflow is very similar: `user.group()` can be used to retrieve the *group name*, and `user.gid()` can be used to retrieve the *group id*. Based on the values these functions return, access can be granted or restricted.

5.3 Which VMODs are shipped with Varnish Enterprise?

Varnish Enterprise ships with a whole bunch of *VMODs*. First of all, all the *Varnish Cache VMODs* are included except `vmod_cookie`. On top of that, the `varnish-modules` *VMOD* collection is added. Then there are the *VMODs* maintained by *Varnish Software*. And finally, there's a collection of *open source VMODs*, which were developed by people in the open source community.

Here's the overview of *VMODs* that are developed and maintained by *Varnish Software*. These modules are exclusively part of *Varnish Enterprise*:

VMOD name	Description
`vmod_accept`	A *content negotiation and sanitization* module that inspects the content of *HTTP request headers*, such as `Accept` and `Accept-Language`
`vmod_aclplus`	An advanced *ACL* module that doesn't require access control information to be explicitly defined in *VCL*, but that can store *ACLs* elsewhere as a *string*
`vmod_akamai`	A module that synchronizes your *Akamai CDN* with *Varnish*
`vmod_brotli`	A *VMOD* that offers *Brotli compression* for *HTTP responses*
`vmod_cookieplus`	An advanced cookie module that allows interacting with both the `Cookie` header on the request side and `Set-Cookie` header on the response side
`vmod_crypto`	A cryptography module
`vmod_deviceatlas`	A *device detection* module that uses the *DeviceAtlas* device intelligence database that matches `User-Agent` information with detailed client-device information
`vmod_edgestash`	A *Mustache*-based templating engine *on the edge* that parses JSON data into Mustache placeholders
`vmod_file`	A module that allows Varnish to interact with the file system, but also act as a fileserver

`vmod_format`	A module for easy string formatting, based on the *ANSI C* `printf` format
`vmod_goto`	A *dynamic backend* module that allows Varnish to connect to non-predefined backends *on-the-fly*
`vmod_headerplus`	Add, remove, update or retrieve any HTTP header
`vmod_http`	A *cURL*-based HTTP client that allows you to perform any HTTP call within *VCL*
`vmod_json`	A *JSON* parsing and introspection module
`vmod_jwt`	Inspect, verify, modify and issue *JSON Web Tokens (JWT)* on the edge
`vmod_kvstore`	A high-performance *in-memory key-value store* with optional TTLs
`vmod_leastconn`	A director module that load balances traffic to the backend with the least number of active connections
`vmod_mmdb`	A *geolocation* module that leverages the *MaxMind GeoIP database* to localize users based on their IP address
`vmod_mse`	Control the *MSE* store selection on a per-request basis
`vmod_resolver`	A module that performs *Forwarded Confirmed reverse DNS (FCrDNS)* on a *client IP*
`vmod_rewrite`	A URL and header rewriting module
`vmod_rtstatus`	A module that presents the *real-time status* of Varnish in *JSON* and *HTML*. Based on internal Varnish counters
`vmod_session`	Control the *idle timeout* of the TCP session
`vmod_sqlite3`	Interact with an *SQLite3* database in *VCL*
`vmod_stale`	A module that implements *Stale If Error* logic to serve stale data if the backend is unhealthy
`vmod_str`	A string manipulation module
`vmod_synthbackend`	Insert *synthetic* objects into the cache as if they were generated by regular backends
`vmod_tls`	Retrieve TLS information from native TLS connections in *Varnish Enterprise*
`vmod_urlplus`	A URL inspection and manipulation module

`vmod_utils`	A collection of utility functions collected in one module
`vmod_waf`	An add-on module that makes Varnish behave like a *Web Application Firewall* by leveraging the *ModSecurity* library
`vmod_xbody`	Access and modify request and response bodies
`vmod_ykey`	A module that performs tag-based invalidation on top of the *MSE stevedore*

Let's have a look at a couple of these enterprise *VMODs*, and see how they add value *on the edge*.

5.3.1 vmod_accept

No need to go in great detail about `vmod_accept` because this *VMOD* was already featured in the previous two chapters. It is used to sanitize the `Accept` and `Accept-Language` headers.

Here's the typical language example:

```
vcl 4.1;

import accept;

sub vcl_init {
    new lang = accept.rule("en");
    lang.add("nl");
}

sub vcl_recv {
    set req.http.Accept-Language = lang.filter(req.http.Accept-Lan-
guage);
}
```

If you want to *Vary* on `Accept-Language`, you need to make sure the number of variations is limited. `vmod_accept` makes sure that this is the case.

Imagine the following `Accept-Language` header:

```
Accept-Language: nl-BE,nl;q=0.9,en-US;q=0.8,en;q=0.7
```

The *VCL example* above will turn this header into the following one:

```
Accept-Language: nl
```

Any other variant will result in the following:

```
Accept-Language: en
```

5.3.2 vmod_aclplus

The typical *ACL* uses the following syntax:

```
acl admin {
    "localhost";
    "secure.my-server.com";
    "192.168.0.0/24";
    ! "192.168.0.25";
}
```

These values are predefined when loading the *VCL file* and offer limited flexibility.

Advanced ACLs

`vmod_aclplus` can load *ACLs* on-the-fly and represents them as a *single-line CSV string*.

This means that *ACLs* can be stored pretty much everywhere:

- In a file
- In a database
- In a key-value store
- In a third-party API

Here's what an *ACL* looks like in this *single-line CSV format*:

```
localhost, secure.my-server.com, 192.168.0.0/24, !192.168.0.25
```

A key-value store example

Here's an example where the *ACL* is stored in a *CSV file* and loaded into the *key-value store* for quick access:

```
vcl 4.1;

import aclplus;
import kvstore;

sub vcl_init {
    new purgers = kvstore.init();
    purgers.init_file("/some/path/data.csv", ",");
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (aclplus.match(client.ip, purgers.get(req.http.host, "er-
ror")) {
            return (purge);
        }
        return (synth(405));
    }
}
```

The *CSV* file for this example is as follows:

```
example.com, localhost, secure.my-server.com, 192.168.0.0/24,
!192.168.0.25
```

And by calling `purgers.init_file("/some/path/data.csv", ",")`, the first part is considered the key, and all the other parts are considered the value.

For requests on the `example.com` hostname, the *ACL* can be loaded. You can actually use arbitrary keys, we just happened to use the `Host` header as the key.

5.3.3 vmod_cookieplus

If you paid attention in *chapter 4*, you might have seen `vmod_cookieplus` used there. This is the example that was used:

```
vcl 4.1;

import cookieplus;

sub vcl_recv {
    cookieplus.keep("language");
    cookieplus.write();
}
```

```
sub vcl_hash {
    if(cookieplus.get("language") ~ "^(nl|en|fr|de|es)$" ) {
        hash_data(cookieplus.get("language"));
    } else (
        hash_data("en");
    }

}
```

This example will remove all cookies, except the *language* cookie. Based on the value of this cookie, language variations are performed.

To some extent, this matches the feature set of `vmod_cookie`, which was added in *Varnish Cache 6.4*. What `vmod_cookie` cannot do, and where `vmod_cookieplus` shines, is controlling cookies that are sent by the backend through the `Set-Cookie` header.

Set-Cookie logic

Here's an example where we will generate a session cookie if it isn't set:

```
vcl 4.1;

import cookieplus;
import crypto;

sub vcl_deliver
{
    set req.http.x-sessid = cookieplus.get("sessid", "");

    if (req.http.x-sessid == "") {
        set req.http.x-sessid = crypto.uuid_v4();
        cookieplus.setcookie_add("sessid", req.http.x-sessid, 30d,
        req.http.Host, "/");
        cookieplus.setcookie_write();
    }
}
```

Unless the `sessid` cookie is set, *Varnish* will set it itself upon delivery. A unique identifier is generated using `crypto.uuid_v4()`. This is what the `Set-Cookie` header will look like when *Varnish* sets it:

```
Set-Cookie: sessid=063f2f1a-3752-43d3-b1e3-fcb2c91f3773;
Expires=Wed, 11 Nov 2020 16:43:15 GMT; Domain=localhost:6081; Path=/
```

5.3.4 vmod_crypto

`vmod_crypto` contains a collection of cryptographic functions that perform hashing, encryption, and encoding. The previous example already contained the `crypto.uuid_v4()` function. Let's have a look at some examples:

Hashing & encoding

```
vcl 4.1;

import crypto;

sub vcl_deliver {
    set resp.http.x-base64 = crypto.base64_encode(crypto.
blob("test"));
    set resp.http.x-md5 = crypto.hex_encode(crypto.hash(md5,"test"));
    set resp.http.x-sha1 = crypto.hex_encode(crypto.
hash(sha1,"test"));
}
```

This example encodes the `test` string in `base64` encoding by leveraging the `crypto.base64_encode()` function. Because this functions takes a `BLOB` argument, conversion using the `crypto.blob()` function is required.

The *VCL snippet* also contains two hashing examples:

- `crypto.hash(md5,"test")` creates an `md5` hash of the `test` string
- `crypto.hash(sh1,"test")` creates a `sha1` hash of the `test` string

In both cases the output is binary and the data type is `BLOB`. That's why the `crypto.hex_encode()` function is required to turn the hashes into strings.

Encryption

Here's the encryption feature of `vmod_crypto` that you already saw in *chapter 2*:

```
vcl 4.1;
import crypto;

sub vcl_recv {
    crypto.aes_key(crypto.blob("my-16-byte-value"));
    return(synth(200, crypto.hex_encode(crypto.aes_encrypt("pass-
word"))));
}
```

The output that is returned by this *VCL* example is `60ed8326cfb1ec02359fff4a-73fe7e0c`. And can be decrypted back into `password` by running the following code: `crypto.aes_decrypt(crypto.hex_decode("60ed8326cfb1ec02359fff4a-73fe7e0c"))`

In *chapter 7* we'll talk about content encryption, and `vmod_crypto` will be used for that.

5.3.5 vmod_deviceatlas

`vmod_deviceatlas` can be used to perform device detection, based on the *DeviceAtlas* dataset. *DeviceAtlas* requires a separate subscription though.

Here's an example in which we will detect mobile users with `vmod_deviceatlas`:

```
vcl 4.1;

import deviceatlas;
import std;

sub vcl_init {
    deviceatlas.loadfile("/etc/varnish/da.json");
}

sub vcl_recv {
    if (deviceatlas.lookup(req.http.User-Agent, "isMobilePhone") ==
"1") {
        std.log("The user-agent is a mobile phone");
    } else if (deviceatlas.lookup(req.http.User-Agent, "isMobile-
Phone") == "0") {
        std.log("The user-agent is not a mobile phone");
    } else if (deviceatlas.lookup(req.http.User-Agent, "isMobile-
Phone") == "[unknown]") {
        std.log("The user-agent is unknown");
    } else {
        std.log("Error during lookup");
    }
}
```

This script logs whether or not the `User-Agent` corresponds with a mobile device.

5.3.5 vmod_edgestash

Mustache is a popular web templating system that uses curly braces to identify placeholders that can be replaced by actual values.

`vmod_edgestash` is *Varnish Enterprise's VMOD* to handle *Mustache syntax*, hence the name *Edgestash*.

This is what your *Mustache template* could look like when the web server returns the HTTP response for `/hello`:

```
Hello {{name}}
```

The `{{name}}` placeholder remains unparsed, and `vmod_edgestash` will pair its value to a *JSON dataset*. This dataset will be loaded via an internal subrequest to `/edgestash.json`.

This *JSON file* could look like this:

```
{
    "name": "Thijs"
}
```

And the following *VCL code* can be used to parse the *Mustache* handlebars, and pair them with *JSON*:

```
vcl 4.1;

import edgestash;

sub vcl_backend_response
{
    if (bereq.url == "/edgestash.json") {
        edgestash.index_json();
    } else if (bereq.url  == "/hello") {
        edgestash.parse_response();
    }
}

sub vcl_deliver
{
    if (req.url == "/hello" && edgestash.is_edgestash()) {
        edgestash.add_json_url("/edgestash.json");
        edgestash.execute();
    }
}
```

The end result will be:

```
Hello Thijs
```

In *chapter 8* we'll have an advanced *Edgestash* example where the dataset is dynamically loaded.

5.3.7 vmod_file

Although `vmod_std` has a `std.fileread()` function to read content from disk, there is still a lot more that can be done with the file system. `vmod_file` offers a broader *API* and some cool features.

File backends

Yes, `vmod_file` can read, write, and delete files. But the coolest feature is the *file backend*.

The following *VCL example* will use the file system as a backend. Whatever is returned from the file system is cached in *Varnish* for subsequent requests:

```
vcl 4.1;

import file;

backend default none;

sub vcl_init {
    new root = file.init("/var/www/html/");
}

sub vcl_backend_fetch {
    set bereq.backend = root.backend();
}
```

If properly configured, file backends can eliminate the need for an actual web server: *Varnish Enterprise* can become the web server.

Command line execution

Another cool `vmod_file` feature is the fact that you can use it to run programs or scripts on the command line.

Here's an integrated example where you can use the custom `UPTIME` request method to retrieve the operating system's uptime:

```
vcl 4.1;

import file;

backend default none;

sub vcl_init {
    new fs = file.init();
    fs.allow("/usr/bin/uptime", "x");
}

sub vcl_recv {
    if (req.method == "UPTIME") {
        return (synth(200, "UPTIME"));
    }
}

sub vcl_synth {
    if (resp.reason == "UPTIME") {
        synthetic(fs.exec("/usr/bin/uptime"));
        if (fs.exec_get_errorcode() != 0) {
            set resp.status = 404;
        }
        return (deliver);
    }
}
```

`fs.exec()` executes the command on the command line. However, there are some significant security issues involved when you allow random scripts and programs to run. That's why the `.exec()` function cannot run unless the `allow_exec` runtime parameter is set. Commands that are to be executed must also be explicitly allowed through a whitelist, the `.allow()` function.

In order for this example to work, you need to run the following command:

```
varnishadm param.set allow_exec true
```

If you want this parameter to be set at runtime, you should add `-p allow_exec=true` to your `varnishd` runtime parameters.

When successfully configured, the uptime service can be called as follows:

```
curl -XUPTIME localhost
```

And the output could be the following:

```
09:53:39 up 2 days, 21:37,  0 users,  load average: 0.79, 0.55, 0.58
```

If the `/usr/bin/uptime` program cannot be successfully called, the service will return an *HTTP 404* status.

Please be careful not to expose too much control or information of your system by using remote execution. Also, please try to limit external access to these commands through an ACL.

5.3.8 vmod_format

`vmod_format` is a *VMOD* that facilitates string formatting, and it does so using the *ANSI C* `printf` format.

You already saw this example in *chapter 2*:

```
vcl 4.1;

import format;

sub vcl_synth {
    set resp.body = format.quick("ERROR: %s\nREASON: %s\n",
        resp.status, resp.reason);
    return (deliver);
}
```

The `format.quick()` function makes it super easy to perform string interpolation. Doing it manually by closing the string, and using the plus-sign can become tedious.

The module also offers functions that allow you to interpolate non-string types. Let's take the previous example and instead of the `format.quick()` function, we'll use the `format.set()`, `format.get()`, `format.add_string()`, and `format.add_int()` functions:

```
vcl 4.1;

import format;

sub vcl_synth {
    format.set("ERROR: %d\nREASON: %s\n");
    format.add_int(resp.status);
    format.add_string(resp.reason);
    set resp.body = format.get();
    return (deliver);
}
```

As you might have noticed, the `%s` format for the numeric status code has been replaced with the actual numeric `%d` format. The `format.add_int()` can now be used to pass in an integer.

When using `format.set()`, the corresponding `format.add_*` functions should be executed in the right order.

5.3.9 vmod_json

The `vmod_json` module can parse *JSON* from a string, or directly from the request body.

The *VCL* example for this *VMOD* will extract the `authorization` property from the JSON object that is represented by the request body:

```
vcl 4.1;

import json;
import std;

sub vcl_recv
{
    std.cache_req_body(100KB);
    json.parse_req_body();

    if (json.is_valid() && json.is_object() &&
            json.get("authorization")) {
        req.http.X-authorization = json.get("authorization");
    } else {
        return(synth(401));
    }
}
```

If the `authorization` property is not set, *Varnish* returns an `HTTP 401 Unauthorized` status code.

5.3.10 vmod_goto

`vmod_goto` is a module that allows you to define backends on the fly. *Varnish Cache* requires you to define all backends upfront. When your backend inventory is dynamic, frequent rewrites and reloads of your *VCL* file are required, which can become cumbersome.

There are also situations where you want to perform one-off calls to an endpoint and cache that data. Or situations where the endpoint is dynamic and based on other criteria.

The DNS backend

Here's an example where the `goto.dns_backend()` method is used to create backends *on the fly*:

```
vcl 4.1;

import goto;

backend default none;

sub vcl_backend_fetch {
    set bereq.backend = goto.dns_backend("backends.example.com");
}
```

This doesn't look very different from a typical backend definition, but internally there is a big difference: regular backends are static and their respective hostnames are resolved *at compile time*. If the *DNS name* changes while a compiled *VCL file* is running, that DNS change will go unnoticed.

Also, traditional backends don't support hostnames that resolve to multiple IP addresses. You'd get an `Backend host "xyz": resolves to too many addresses` error.

However, `vmod_goto` can handle these types of *A-records*, and will cycle through the corresponding IP addresses at runtime. If at some point your origin architecture needs to scale, `vmod_goto` can handle *DNS changes* without reloading the *VCL file*.

You can even use the *URL* as an endpoint:

```
vcl 4.1;

import goto;

backend default none;

sub vcl_backend_fetch {
    set bereq.backend = goto.dns_backend("https://backends.example.
com:8888");
}
```

This example will connect to `backends.example.com`, and if multiple IP addresses are assigned to that hostname, it will cycle through them. The connection will be made using *HTTPS* over port `8888`.

The DNS director

Here's an example where the `goto.dns_director()` method is used to create a director object that resolves hostnames dynamically:

```
vcl 4.1;

import goto;

backend default none;

sub vcl_init {
    new dyndir = goto.dns_director("backends.example.com");
}

sub vcl_backend_fetch {
    set bereq.backend = dyndir.backend();
}
```

Extra options

Both `goto.dns_backend()` and `goto.dns_director()` have a bunch of extra optional arguments that allow you to fine-tune some of the connection parameters. They are on par with the capabilities of regular backends, but they also have *TTL settings* to control the *DNS resolution*.

Because the arguments are named, it doesn't really matter in which order you use them. Here's an example:

```
vcl 4.1;

import goto;

backend default none;

sub vcl_backend_fetch {
    set bereq.backend = goto.dns_backend("https://backends.example.
com:8888",
        host_header="example.com", connect_timeout=2s, first_byte_
timeout=10s,
        max_connections=10, ssl_verify_peer=false, ip_version=ipv4,
        ttl=1h, ttl_rule=morethan);
}
```

This does the following:

- It connects to `backends.example.com`
- The connection is made over *HTTPS* on port `8888`
- The `Host` header that is used for the request will be overridden to `example.com`
- The *connection timeout* is *two seconds*
- the *first byte timeout* is *ten seconds*
- We allow *ten simultaneous connections* to the backend
- Although *TLS/SSL* is used for the connection, *peer verification* is disabled
- When we resolve the hostname, we only care about *IPv4 addresses*
- The *DNS TTL* from the record itself will be used, as long as it is more than the `ttl` value we set

Dynamic backends example

Although `vmod_goto` is considered a *dynamic backend VMOD*, the examples looked a bit static. The real dynamic behavior is behind the scenes, but it's not very inspiring.

Let's throw in a cool example where the origin inventory is stored in a *JSON* file.

Imagine every tenant having its own backend configuration file that is located in `/etc/varnish/tenants/xyz.json`, where the `xyz` refers to the hostname of the tenant.

In our case there is a `/etc/varnish/tenants/example.com.json` file that has the follow content:

```
{
  "origin": {
    "addr": "https://backends.example.com",
    "connect": "1s"
  }
}
```

By leveraging `vmod_goto`, `vmod_file`, `vmod_std`, and `vmod_json`, we can read and parse the file, extract the parameters, and feed them to `vmod_goto`:

```
vcl 4.1;

import goto;
import file;
import std;
import json;

backend default none;

sub vcl_init {
    new fs = file.init();
    fs.allow("/etc/varnish/tenants/*.json");
}

sub vcl_backend_fetch {
    json.parse(fs.read("/etc/varnish/tenants/" + bereq.http.host +
".json"));

    if (!json.get("origin.addr")) {
        return(abandon);
    }

    set bereq.backend = goto.dns_backend(json.get("origin.addr"),
    connect_timeout = std.duration(json.get("origin.connect"),
3.5s));
}
```

Based on `/etc/varnish/tenants/example.com.json`, the backend would be `https://backends.example.com` with a *connection timeout* of *one second*.

5.3.11 vmod_headerplus

`vmod_headerplus` was already covered when we discussed the new features of *Varnish Enterprise 6* in *chapter 2*. Let's make sure we use a different *VCL example* to show the power of this *VMOD*:

The following example will add *stale-if-error* support, which *Varnish* doesn't support by default:

```
vcl 4.1;
import headerplus;

sub vcl_backend_response {
    headerplus.init(beresp);
    set beresp.http.stale-if-error =
    headerplus.attr_get("Cache-Control", "stale-if-error");
    if (beresp.http.stale-if-error != "") {
        set beresp.grace = std.duration(bereq.http.stale-if-error +
"s", 11s);
    }
    unset beresp.http.stale-if-error;
}
```

The `headerplus.attr_get()` function allows us to retrieve specific attributes from a header. In this case we're retrieving the `stale-if-error` attribute. The value of this attribute is assigned to `beresp.grace`, which sets the *grace mode*.

5.3.12 vmod_http

Being able to perform arbitrary *HTTP calls* from within *VCL* is a very powerful concept. `vmod_http` offers us the tools to do this. This module wraps around `libcurl`, which is an *HTTP client library*.

There are many ways in which `vmod_http` can be leveraged, but in the example we'll use feature *content prefetching*:

```
vcl 4.1;

import http;

sub vcl_recv {
    set req.http.X-prefetch = http.varnish_url("/");
}
```

```
sub vcl_backend_response {
    if (beresp.http.Link ~ "<.+>; rel=prefetch") {
        set bereq.http.X-link = regsub(beresp.http.Link,
"^.*<([^>]*)>.*$", "\1");
        set bereq.http.X-prefetch = regsub(bereq.http.X-prefetch,
"/$", bereq.http.X-link);

        http.init(0);
        http.req_copy_headers(0);
        http.req_set_method(0, "HEAD");
        http.req_set_url(0, bereq.http.X-prefetch);
        http.req_send_and_finish(0);
    }
}
```

This example will prefetch a URL that was mentioned in the `Link` header.

Imagine the following `Link` response header:

```
Link: </style.css>; rel=prefetch
```

The *VCL* example above will match the pattern of this header and use `regsub` to extract the URL, concatenate it with the full Varnish URL, and call this URL in the background.

The `http.req_send_and_finish(0)` function call ensures that the HTTP request is made in the background, and that we're not waiting for the response. The fact that the `HEAD` request method is used also means we don't need to process the response body. Subrequests that are sent to *Varnish* using `HEAD` will automatically be converted into a full `GET` request. This means the response body will be stored in cache.

When `/style.css` is eventually called by the client, it is already stored in cache, ready to be served.

5.3.13 vmod_jwt

`vmod_jwt` is a module for verifying, creating, and manipulating *JSON Web Tokens* and *JSON Web Signatures*. It was already covered in *chapter 2*, when we talked about new features in *Varnish Enterprise 6*.

Here's one example we had:

```
vcl 4.1;

import jwt;

sub vcl_init {
    new jwt_reader = jwt.reader();
}

sub vcl_recv {
    if (!jwt_reader.parse(regsub(req.http.Authorization,"^Bearer
",""))) {
        return (synth(401, "Invalid JWT Token"));
    }

    if (!jwt_reader.set_key("secret")) {
        return (synth(401, "Invalid JWT Token"));
    }

    if (!jwt_reader.verify("HS256")) {
        return (synth(401, "Invalid JWT Token"));
    }
}
```

This is what one possible *JWT* could look like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM-
5MDIyfQ.
XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

It consists of three distinct parts, separated by a dot, encoded with `base64` This is what the decoded version looks like:

The first part is the header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The second part is the actual payload of the token:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

The third part is the signature, which contains binary data, and is secured by the secret key.

The *JWT* would be used as follows:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM-
5MDIyfQ.
XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

Our *VCL example* would verify the token based on the format, but the `.verify()` method would ensure the *HMAC signature* matches.

In *chapter 8* we'll talk more about authentication, and we'll feature some of the capabilities of `vmod_jwt`.

5.3.14 vmod_kvstore

`vmod_kvstore` offers a built-in *key-value store*. This is a simple *memory database* that stores keys with their associated value. The key name is always a string; the value is also a string. By default `vmod_kvstore` has a global state, so values remain available across requests.

The *KVStore* has the typical *getters, setters, and counters*, but it can also be populated at startup from a file.

`vmod_kvstore` was also part of the example we used for `vmod_aclplus`. Let's focus on the *KVStore* functionality of the example:

```
vcl 4.1;

import aclplus;
import kvstore;

sub vcl_init {
    new purgers = kvstore.init();
    purgers.init_file("/some/path/data.csv", ",");
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (aclplus.match(client.ip, purgers.get(req.http.host, "er-
ror")) {
            return (purge);
        }
        return (synth(405));
    }
}
```

The `.init_file("/some/path/data.csv", ",")` will read `/some/path/data.csv` and extract the values, using `,` as a separator. The first value will be used as the key, the rest is considered the value. The `.get()` method will fetch the key that was passed, and will return the corresponding value.

5.3.15 vmod_mmdb

The `mmdb` in `vmod_mmdb` is short for *MaxMind DB*. MaxMind is a company that provides *geoIP databases*. This *VMOD* is a wrapper around `libmaxminddb`, which contains the *API* to access and parse the *geoIP* database.

This *VMOD* was also part of the new features of *Varnish Enterprise 6*, and was covered in *chapter 2*. Here's the example we used there:

```
vcl 4.1;

import mmdb;

sub vcl_init {
    new geodb = mmdb.init("/path/to/database");
}

sub vcl_recv {
    return(synth(200,
        "Country: " + geodb.lookup(client.ip, "country/names/en") +
" - " +
        "City: " + geodb.lookup(client.ip, "city/names/en")
    ));
}
```

The `mmdb.init()` function will fetch the *geoIP* data from a datafile, and results in an object that contains lookup functionality.

In this example `geodb.lookup(client.ip, "country/names/en")` is used to extract the country name from the *geoIP data* that is associated with the *client IP address*.

5.3.16 vmod_mse

The *Massive Storage Engine* is one of the key features of *Varnish Enterprise*. As mentioned before, it is a *stevedore* that combines memory and file-backed storage.

MSE's persistence layer consists of *books*, which are metadatabases, and *stores*, which are the underlying datastores. Books and stores can be tagged, and `vmod_mse` has the ability to select a tagged store, and as a result choose where objects get stored.

In *chapter 2*, `vmod_mse` was covered when we talked about new features in *Varnish Enterprise*. The internal configuration of *MSE* was also showcased.

Here's an example that was used in that section to illustrate how *tags* in *MSE* can be used to select where objects get stored:

```
vcl 4.1;

import mse;
import std;

sub vcl_backend_response {
    if (beresp.ttl < 120s) {
        mse.set_stores("none");
    } else {
        if (beresp.http.Transfer-Encoding ~ "chunked" ||
        std.integer(beresp.http.Content-Length,0) > std.bytes("1M"))
{
            mse.set_stores("sata");
        } else {
            mse.set_stores("ssd");
        }
    }
}
```

In this case, there are two types of *MSE stores*: stores that are tagged with `sata`, and stores that are tagged with `ssd`. These tags reflect the type of disks that are used for these *books*.

The *VCL* example forces objects to be stored on `sata` tagged books, if the `Content-Length` header indicates a size bigger than *1 MB*, or if we don't have a `Content-Length` header at all, and `Transfer-Encoding: chunked` is used instead.

Please note that it is not recommended to use *MSE* with spinning disks on servers with high load.

5.3.17 vmod_resolver

`vmod_resolver` is a module that performs *Forwarded Confirmed reverse DNS (FCrDNS)*. This means that a *reverse DNS* call is done on the *client IP address*.

The resulting hostname is then resolved again, and if this matches the original *client IP address*, the check succeeds.

Here's the reference example:

```
vcl 4.1;

import std;
import resolver;

sub vcl_recv {
    if (resolver.resolve()) {
        std.log("Resolver domain: " + resolver.domain());
    } else {
         std.log("Resolver error: " + resolver.error());
     }
}
```

`resolver.resolve()` performs the *FCrDNS* and returns `true` when it succeeds. In that case the `resolver.domain()` returns the domain that came out of this. When the resolver fails, we can use `resolver.error()` to retrieve the error message.

5.3.18 vmod_rewrite

URL matching and rewriting is a very common practice in *VCL*. The language itself offers us the syntax to do this. But the more logic is added, the harder it gets to manage the rules, and the bigger the complexity. It's basically like any other software project.

Rewrite rules in VCL

Here's an example of what it could look like:

```
vcl 4.1;

sub vcl_recv {
    if (req.url ~ "/pay") {
        set req.url = regsub(req.url, "/pay", "/checkout");
    } else if (req.url ~ "(?i)/cart") {
        set req.url = regsub(req.url, "(?i)/cart", "/shopping-cart");
    } else if (req.url ~ "product\-([0-9]+)") {
        set req.url = regsub(req.url, "product\-([0-9]+)", "cata-
log\-\1");
    }
}
```

You have to admit that this workflow gets messy really quickly. However, `vmod_rewrite` helps you organize your URL rewriting logic through rules.

vmod_rewrite rulesets

You can create a *rules file* that contains all the logic. Here's what it can look like:

```
"/pay"               "/checkout"
"(?i)/cart"          "/shopping-cart"
"product-([0-9]+)" "catalog-\1"
```

This file can then be loaded into *VCL* using the following code:

```
vcl 4.1;

import rewrite;

sub vcl_init {
    new rs = rewrite.ruleset("/path/to/file.rules");
}

sub vcl_recv {
    set req.url = rs.replace(req.url);
}
```

And that's all it takes. The logic is separated in a different file, it doesn't pollute your *VCL*, and is easy to manage.

Rulesets as a string

The same logic can also be loaded as a string in the *VCL* without compromising too much on readability:

```
vcl 4.1;

import rewrite;

sub vcl_init {
    new rs = rewrite.ruleset(string = {"
        "/pay"                "/checkout"
        "(?i)/cart"           "/shopping-cart"
        "product-([0-9]+)" "catalog-\1"
    "});
}

sub vcl_recv {
    set req.url = rs.replace(req.url);
}
```

Matching URL patterns

So far we've been rewriting *URLs*, but we can also match patterns using the `.match()` method. Here's a quick example:

```
vcl 4.1;

import rewrite;

sub vcl_init {
    new rs = rewrite.ruleset(string = {"
        "^/admin/"
        "^/purge/"
        "^/private"
    "}, min_fields = 1);
}

sub vcl_recv {
    if (rs.match(req.url)) {
        return (synth(405, "Restricted");
    }
}
```

The *ruleset* contains a list of private *URLs* that cannot be accessed through *Varnish*. `rs.match(req.url)` checks whether or not the *URL* matches those rules.

Extracting ruleset fields

The final example uses *rewrite logic*, not necessarily to rewrite the *URL*, but to extract values from a field in the ruleset.

Here's the code:

```
vcl 4.1;
import std;
import rewrite;

sub vcl_init {
    new rs = rewrite.ruleset(string = {"
        # pattern        ttl     grace   keep
        "\.(js|css)"     "1m"    "10m"   "1d"
        "\.(jpg|png)"    "1w"    "1w"    "10w"
    "});
}

sub vcl_backend_response {
    # if there's a match, convert text to duration
    if (rs.match(bereq.url)) {
        set beresp.ttl   = std.duration(rs.rewrite(0, mode = only_
matching), 0s);
        set beresp.grace = std.duration(rs.rewrite(1, mode = only_
matching), 0s);
        set beresp.keep  = std.duration(rs.rewrite(2, mode = only_
matching), 0s);
    }
}
```

If a *backend request's URL* matches our ruleset, fields are extracted that represent the corresponding *TTL*, *grace time*, and *keep time*.

If your *VCL* has specific logic to assign custom *TTL* values to certain *URL* patterns, `vmod_rewrite` can take care of this for you.

5.3.19 vmod_sqlite3

As mentioned in *chapter 2* when talking about new features in *Varnish Enterprise 6*: *SQLite* is a library that implements a serverless, self-contained relational database system.

`vmod_sqlite3` wraps around this library and allows you to write *SQL statements* to interact with *SQLite*. The data itself is stored in a single file, as you can see in the example below:

```
vcl 4.1;

import sqlite3;
import cookieplus;
backend default none;

sub vcl_init {
    sqlite3.open("sqlite.db", "|;");
}

sub vcl_fini {
    sqlite3.close();
}

sub vcl_recv {
    cookieplus.keep("id");
    cookieplus.write();
    if(cookieplus.get("id") ~ "^[0-9]+$") {
        set req.http.userid = cookieplus.get("id");
        set req.http.username = sqlite3.exec("SELECT `name` FROM `us-
ers`
        WHERE rowid=" + sqlite3.escape(req.http.userid));
    }
    if(!req.http.username || req.http.username == "") {
        set req.http.username = "guest";
    }
    return(synth(200,"Welcome " + req.http.username));
}
```

This example will fetch the *user's id* from the `id` cookie and fetch the corresponding row from the database. When there's a match, the username is returned. Otherwise *guest* is returned as a value.

5.3.20 vmod_stale

`vmod_stale` is also one of the new *VMODS* covered in *chapter 2*. It is a way to revive stale objects that are about to expire.

Varnish has cascading timers to keep track of an object's lifetime:

- Objects whose *TTL* hasn't expired are considered *fresh*.
- When the *TTL* of an object has expired, but there is still *grace* left, the object is considered *stale*.
- When even the *grace* has expired, an object can be kept around if there is *keep* time left.

As long as the object is still around `vmod_stale` can reset its *TTL, grace, and keep value*.

Here's the example that we used in *chapter 2*:

```
vcl 4.1;

import stale;

sub stale_if_error {
    if (beresp.status >= 500 && stale.exists()) {
        stale.revive(20m, 1h);
        stale.deliver();
        return (abandon);
    }
}

sub vcl_backend_response {
    set beresp.keep = 1d;
    call stale_if_error;
}

sub vcl_backend_error {
    call stale_if_error;
}
```

By setting `beresp.keep` to one day, we make sure the object is kept around long enough, even though its *TTL and grace* have expired. This allows `std.revive()` to revive the object and make it fresh again.

In this case, the object is *fresh* for another 20 minutes, and after that an hour of *grace* is added so *Varnish* can perform a *background fetch* for revalidation while the *stale* is served.

This *revival* example only happens when the *origin server* is responding with *HTTP 500*-style errors. This is basically a `stale-if-error` implementation.

5.3.21 vmod_synthbackend

By now, you should be quite familiar with *synthetic output*. But if you return `synth()` in *VCL*, the response happens *on the fly*, and is not cached.

As explained in *chapter 2*, `vmod_synthbackend` will allow you to define a *synthetic backend* that caches the *synthetic output*.

Here's a code example that illustrates how this can be done:

```
vcl 4.1;

import synthbackend;

backend default none;

sub vcl_backend_fetch {
    set bereq.backend = synthbackend.from_string("URL: " + bereq.url
+ ", time: " + now);
}
```

In this case, the *synthetic output* will be stored in cache for the duration of the `default_ttl` runtime parameter. By default this is two minutes.

5.3.22 vmod_tls

In the previous section about *VMODs* that are shipped with *Varnish Cache*, we talked about `vmod_proxy` that retrieves *TLS information* from a *PROXY protocol* connection.

Varnish Enterprise offers native TLS support, and when activated, no *PROXY* connection is needed. In order to support the same functionality as `vmod_proxy` for local TLS connections, we developed `vmod_tls`.

The *API* is identical; it's just a different module. Here's the same example as for `vmod_proxy`, but using `vmod_tls`:

```
vcl 4.1;
import tls;

sub vcl_deliver {
    set resp.http.alpn = tls.alpn();
    set resp.http.authority = tls.authority();
    set resp.http.ssl = tls.is_ssl();
    set resp.http.ssl-version = tls.ssl_version();
    set resp.http.ssl-cipher = tls.ssl_cipher();
}
```

And this is some example output, containing the custom headers:

```
alpn: h2
authority: example.com
ssl: true
ssl-version: TLSv1.3
ssl-cipher: TLS_AES_256_GCM_SHA384
```

5.3.23 vmod_urlplus

`vmod_urlplus` was extensively covered in the *New features in Varnish Enterprise 6* section of *chapter 2*.

We'll just throw in one of the examples to refresh your memory:

```
vcl 4.1;

import urlplus;

sub vcl_recv
{
    //Remove all Google Analytics
    urlplus.query_delete_regex("utm_");

    //Sort query string and write URL out to req.url
    urlplus.write();
}
```

This example will remove all *query string parameters* that match the `utm_` pattern from the *URL*. This makes a lot of sense, because these parameters add no value from a caching point of view. On the contrary, they cause too much variation to be created and have an detrimental impact on the hit rate.

5.3.24 vmod_xbody

`vmod_xbody` is a module that provides access to request and response bodies. It is also capable of modifying the request and response bodies.

Imagine having the following static content on your website:

```
Hello Thijs
```

Using `vmod_xbody`, and some other *VMODs*, you can replace the name with any name you want.

The following example uses the *KVStore* to store various names, identified by an *ID*. The name that is displayed depends on the value of the `id` cookie. The corresponding name is retrieved from the *KVStore*:

```
vcl 4.1;

import xbody;
import kvstore;
import cookieplus;

sub vcl_init {
    new people = kvstore.init();
    people.set("1","Thijs");
    people.set("2","Lex");
    people.set("3","Lize");
    people.set("4","Lia");
}

sub vcl_backend_response {
    xbody.regsub("Thijs", people.get(cookieplus.
get("id","0"),"guest"));
}
```

So if the cookie header is `Cookie: id=2`, the output will be as follows:

```
Hello Lex
```

If the cookie header is `Cookie: id=4`, the output will be:

```
Hello Lia
```

If the `id` cookie is not set, or has an unknown value, the output will be:

```
Hello guest
```

5.3.25 vmod_ykey

`vmod_ykey` is the *Varnish Enterprise* version of `vmod_xkey`. The naming is a bit funny, we know.

`vmod_xkey` is an open source module that adds secondary keys to objects. This allows us to purge objects from cache based on tags rather than the *URL*. Unfortunately `vmod_xkey` proved to be incompatible with *MSE*, so `vmod_ykey` was built to tackle this issue.

As this *VMOD* was already featured in *chapter 2*, we'll just show you the example:

```
vcl 4.1;

import ykey;

acl purgers { "127.0.0.1"; }

sub vcl_recv {
    if (req.method == "PURGE") {
        if (client.ip !~ purgers) {
            return (synth(403, "Forbidden"));
        }
        set req.http.n-gone = ykey.purge_header(req.http.Ykey-Purge,
sep=" ");
        return (synth(200, "Invalidated "+req.http.n-gone+" ob-
jects"));
    }
}

sub vcl_backend_response {
    ykey.add_header(beresp.http.Ykey);
    if (bereq.url ~ "^/content/image/") {
        ykey.add_key("image");
    }
}
```

An application can add tags by stating them in the `Ykey` response header, which this *VCL* script can parse. Content for which the *URL* matches the `/content/image/` pattern will automatically have an `image` tag assigned to it.

By invalidating a tag, all tagged objects are removed from cache at once.

You just need to send the following *HTTP request* to *Varnish* if you want to remove all objects that are tagged with the `image` tag:

```
PURGE / HTTP/1.1
Ykey-Purge: image
```

The next chapter is all about cache invalidation. We'll talk about `vmod_ykey` in a lot more detail there.

5.4 Where can you find other VMODs?

The *Varnish Cache* core contributors and the *Varnish Software* development team aren't the only ones building *VMODs*.

The *Varnish Cache* website has a section dedicated to *third-party VMODs*: http://varnish-cache.org/vmods/.

You can also search on *GitHub* for repositories that start with `libvmod-`. Just have a look at the following results: https://github.com/search?q=libvmod-&type=repositories.

There are some really interesting ones there, but before you install one, make sure it is compatible with *Varnish 6*.

5.4.1 Third-party VMODs

There are a lot of good third-party *VMODs* out there, but there's also some overlap with functionality that can be found in *Varnish Enterprise VMODs*.

Here are two third-party *VMODS* I really like:

- `vmod_basicauth`
- `vmod_redis`

vmod_basicauth

The `vmod_basicauth` module reads a typical *Apache* `.htpasswd` file and tries to match it to the incoming `Authorization` header.

Here's an example of this *VMOD*:

```
vcl 4.1;

import basicauth;

sub vcl_recv {
    if (!basicauth.match("/var/www/.htpasswd",req.http.Authoriza-
tion)) {
        return (synth(401, "Restricted"));
    }
}
```

```
sub vcl_synth {
    if (resp.status == 401) {
        set resp.http.WWW-Authenticate = {"Basic realm="Restricted
area""};
    }
}
```

If the *username and password* encoded in the `Authorization` header don't match an entry in `.htpasswd`, an *HTTP 401* status code is triggered, which will result in a `WWW-Authenticate: Basic realm="Restricted area"` response header being synthetically returned.

This `WWW-Authenticate` header will trigger a *server-side login popup*.

For more information about this *VMOD*, please visit http://man.gnu.org.ua/manpage/?3+vmod-basicauth.

vmod_redis

`vmod_redis` provides a *client API* to control a *Redis* server. Redis is an advanced *distributed key-value store* and has become somewhat of an industry standard.

Here's just a very simple example where we set up a connection and fetch the value of the `foo` key:

```
vcl 4.1;

import redis;

sub vcl_init {
    new db = redis.db(
        location="192.168.1.100:6379",
        connection_timeout=500,
        shared_connections=false,
        max_connections=2);
}

sub vcl_deliver {
    db.command("GET");
    db.push("foo");
    db.execute();
    set resp.http.X-Foo = db.get_string_reply();
}
```

But the *API* for this *VMOD* is extensive and supports some of the following features:

- Working with *clustered and replicated* setups
- Running *LUA* scripts
- Evaluating various return types
- Request pipelining
- Leveraging the various *Redis* data types

For more information about this *VMOD*, please visit https://github.com/carlosabalde/libvmod-redis.

5.4.2 The Varnish Software VMOD collection

Varnish Software also has a GitHub repository with some open source *VMODs*. You can find the code on https://github.com/varnish/varnish-modules.

These modules are also packaged with *Varnish Enterprise*. Here is the list of *VMODs* that are part of this collection:

- `vmod_bodyaccess`
- `vmod_header`
- `vmod_saintmode`
- `vmod_tcp`
- `vmod_var`
- `vmod_vsthrottle`
- `vmod_xkey`

Let's do a walkthrough, and look at some *VCL examples* for a couple of these *VMODs*.

vmod_bodyaccess

`vmod_bodyaccess` is a very limited version of `vmod_xbody`. Whereas `vmod_xbody` has read and write access to request and response bodies, `vmod_bodyaccess` only has read access to the request body.

Here's a list of features that this *VMOD* provides:

- Search for *regular expression pattern* in the request body
- Hash the request body
- Get the request body length
- Log the request body to *VSL*

Let's have a look at a *VCL example* where we'll use `vmod_bodyaccess` in a scenario where we'll cache `POST` requests:

```
vcl 4.1;

import std;
import bodyaccess;

sub vcl_recv {
    set req.http.x-method = req.method;
    if (req.method == "POST") {
        if (std.cache_req_body(110KB)) {
            if (bodyaccess.rematch_req_body("id=[0-9]+")) {
                return (hash);
            }
            return (synth(422, "Missing ID");
        }
        return (synth(413));
    }
}

sub vcl_hash {
    bodyaccess.hash_req_body();
}

sub vcl_backend_fetch {
    set bereq.method = bereq.http.x-method;
}
```

None of this works unless `std.cache_req_body()` is called. This starts caching the request body. `bodyaccess.rematch_req_body("id=[0-9]+")` is used to figure out whether or not the `id=[0-9]+` pattern is part of the request body. When the pattern matches, we'll decide to cache. If the payload is too large or the ID is missing an error will be generated instead.

`bodyaccess.hash_req_body()` is used in `vcl_hash` to create a cache variation for each request body value.

This is the *HTTP* request you can send to trigger this behavior:

```
POST / HTTP/1.1
Host: localhost
Content-Length: 4
Content-Type: application/x-www-form-urlencoded

id=1
```

vmod_header

`vmod_header` allows you to get headers, append values to headers, and remove values from headers. It's a very limited *VMOD* in terms of functionality, and pales in comparison to `vmod_headerplus`.

Here's a quick example, just for the sake of it:

```
vcl 4.1;

sub vcl_backend_response {
    header.remove(beresp.http.foo, "one=1");
}
```

Imagine the following *HTTP response headers*:

```
HTTP/1.1 200 OK
Foo: one=1
Foo: one=2
```

The response contains two instances of the `Foo` header, each with different values. The example above will ensure that the first occurrence of the header that matches the pattern is removed.

`Foo: one=1` matches that pattern and is removed. Only `Foo: one=2` remains.

vmod_tcp

The *TCP VMOD* allows you to control certain aspects of the underlying *TCP connection* that was established.

The first example will enable rate limiting for incoming connections:

```
vcl 4.1;

import tcp;

sub vcl_recv
{
    tcp.set_socket_pace(1024);
}
```

This example will pace the throughput at a rate of *1024 KB/s*.

In the second example the `BBR` congestion algorithm is used for *TCP connections*:

```
vcl 4.1;

import tcp;

sub vcl_recv {
    tcp.congestion_algorithm("bbr");
}
```

This requires that the `BBR` congestion controller is both available and loaded.

vmod_var

VCL clearly lacks the concept of *variables*. Although headers are commonly used to transport values, the values are usually cast to strings, and one needs to have the discipline to strip off these headers before delivering them to the backend or the client.

`vmod_var` offers variable support for various data types.

The supported data types are:

- strings
- integers
- real numbers
- durations
- IP addresses
- backends

Here's the *VCL example* to illustrate some of the *VMOD's* functions:

```
vcl 4.1;

import var;
import std;

sub vcl_recv {
    var.set_ip("forwarded",std.ip(req.http.X-Forward-
ed-For,"0.0.0.0"));
    var.set_real("start",std.time2real(now,0.0));
}

sub vcl_deliver {
    set resp.http.forwarded-ip = var.get_ip("forwarded");
    set resp.http.start = std.real2time(var.get_real("start"),now);
}
```

This example will use the `std.ip()` function from `vmod_std` to turn the `X-Forwarded-For` header into a valid `IP` type. The value is stored using `var.set_ip()`.

The current time is also stored as a `real` type, using `std.time2real()` to convert the type.

In a later stage, we can retrieve the information using the corresponding *getter functions*.

vmod_vsthrottle

We've already seen *rate limiting* when we talked about `vmod_tcp`. But rate limiting also happens in this *VMOD*.

In `vmod_vsthrottle`, we're restricting the number of requests a client can send in a given timeframe.

Take for example, the *VCL code* below:

```
vcl 4.1;

import vsthrottle;

sub vcl_recv {
    if (vsthrottle.is_denied(client.identity, 15, 10s, 30s)) {
        return (synth(429, "Too Many Requests. You can retry in "
        + vsthrottle.blocked(client.identity, 15, 10s, 30s)
        + " seconds."));
    }
}
```

This code will only allow clients to perform *15 requests in a 10 second timeframe*. If that rate is exceeded, the user gets blocked for *30 seconds*.

The `vsthrottle.is_denied()` function is responsible for that. The `vsthrottle.blocked()` is also quite helpful, as it returns the number of seconds the user is still blocked.

This allows us to set expectations. This is the error messages that users get when they are blocked due to rate limiting:

```
Too Many Requests. You can retry in 26.873 seconds.
```

In this case, the user knows they should wait another *26 seconds* before attempting to perform another request.

vmod_xkey

We already mentioned that `vmod_xkey` is the predecessor of `vmod_ykey`, basically the open source version. It doesn't work when using *MSE*, but if you're using *Varnish Cache*, that is not a concern.

The *API* is also more limited, as you are forced to tag objects using the `xkey` response header, and you cannot add extra tags in *VCL*.

Here's the *VCL example* to show how you can remove objects from cache using `vmod_xkey`:

```
vcl 4.1;

import xkey;

sub vcl_recv {
    if (req.method == "PURGE" && req.http.x-xkey-purge) {
        if (xkey.purge(req.http.x-xkey-purge) != 0) {
            return(synth(200, "Purged"));
        }
        return(synth(404, "Key not found"));
    }
}
```

If you're performing a *PURGE* call, you can use the `x-xkey-purge` request header to specify the keys you want to use for purging. The keys are *space-delimited*.

Imagine we want to remove all the objects that are tagged with the `js` and `css` tags. You'd need to send the following *HTTP request*:

```
PURGE / HTTP/1.1
x-xkey-purge: js css
```

The response would be `HTTP/1.1 200 Purged` if the keys were found or `HTTP/1.1 404 Key not found` when there are no corresponding keys.

Please note that you usually want to limit such functionality behind ACL. In the next chapter, we'll talk about *cache invalidation*, and we'll cover the `vmod_xkey` in more detail as well.

5.4.3 How to install these VMODs

We always advise you to install *Varnish Cache* using our official packages. These are available on https://packagecloud.io/varnishcache. However, we don't provide packages for the *VMOD collection*. This means you'll have to compile them from source.

Compiling from source

You can get the source code from https://github.com/varnish/varnish-modules. But for *Varnish Cache 6.0* and *Varnish Cache 6.0 LTS*, you need to download the code from the right branch:

- For *Varnish Cache 6.0*, the source can be downloaded from https://github.com/varnish/varnish-modules/archive/6.0.zip.
- For *Varnish Cache 6.0 LTS*, the source can be downloaded from https://github.com/varnish/varnish-modules/archive/6.0-lts.zip.

There are some build dependencies that need to be installed. On a *Debian* or *Ubuntu* systems, you can install them using the following command:

```
# apt-get install -y varnish-dev autoconf automake gcc libtool make
python3-docutils
```

On *Red Hat*, *Fedora*, and *CentOS* systems, you can use the following command to install the dependencies:

```
# yum install -y varnish-devel autoconf automake gcc libtool make py-
thon3-docutils
```

Once all build dependencies are in place, you can compile the *VMODs* by running the following commands in the directory where the *VMOD* source files were extracted:

```
./bootstrap
./configure
make
make install
```

After having run `make install`, the corresponding `.so` files for these *VMODs* can be found in the path that was defined by the `vmod_dir` runtime parameter in *Varnish*.

Debian and Ubuntu distro packages

The *Debian* and *Ubuntu* distributions also offer *Varnish Cache* via their own packages. Some versions still provide the `varnish-modules` package, which is *Debian and Ubuntu's* version of the *VMOD collection*.

Installing these *VMODS* is done using this very simple command:

```
# apt-get install varnish-modules
```

The following version of *Debian* and *Ubuntu* offer the `varnish-modules` package for *Varnish Cache 6*:

- *Ubuntu 20.10 (Groovy)* offers *Varnish Cache 6.4.0*.
- *Debian 10 (Buster)* offers *Varnish Cache 6.1.1*.
- *Debian 11 (Bullseye)* offers *Varnish Cache 6.4.0*.

Although we advise installing *Varnish Cache 6.0 LTS* from our official packages, there's no denying that it's easy to just install *Varnish* using a simple `apt-get install varnish` command.

5.5 Writing your own VMODs

Anything that can be written in `C` can become a *VMOD*. In this chapter we've gone through a long list of *VMODs*: Some of them are managed by the *Varnish Cache* team, a lot of them are managed by *Varnish Software*, and then there are *VMODs* that are managed by individual contributors.

A lot of common scenarios are already covered by a *VMOD*, but there's always a chance that you have a use case where *VCL* cannot solve the issue, and there isn't a matching *VMOD* either.

In that case, you can write one yourself. That's what we're going to do in this section. Even if you're not planning to write your own *VMOD*, it is still interesting to learn how *VMODs* are composed.

5.5.1 vmod_example

To get started with *VMOD* development, you should have a look at https://github.com/varnishcache/libvmod-example. This GitHub repository hosts the code and build scripts for `vmod_example`.

`vmod_example` is a stripped down *VMOD* that serves as the boilerplate. It is the ideal starting point for novice *VMOD* developers.

This is what the directory structure of this repo looks like:

```
.
|-- Makefile.am
|-- autogen.sh
|-- configure.ac
|-- m4
|   `-- ax_pthread.m4
|-- rename-vmod-script
`-- src
    |-- Makefile.am
    |-- tests
    |   `-- test01.vtc
    |-- vmod_example.c
    `-- vmod_example.vcc
```

The core of the code is the `src` folder where the source files are located:

- `vmod_example.c` contains the source code of this *VMOD*.
- `vmod_example.vcc` contains the interface between the code and the *VCL compiler (VCC)*.

A useful script is `rename-vmod-script`: you're not going to name your custom *VMOD* `vmod_example`. This script is there to rename the *VMOD* and replace all the occurrences of `example` with the actual name of the new *VMOD*.

So, if you wanted to name your *VMOD* `vmod_os`, you'd do the following:

```
./rename-vmod-script os
```

This means `vmod_example` is renamed to `vmod_os`.

The *VMOD* we're going to develop will display operating system information, hence the name `vmod_os`.

Files like `Makefile.am`, `configure.ac`, and the `m4` directory are there to facilitate the build process. They are used by `autoconf`, `automake`, and `libtool`, and are triggered by the `autogen.sh` shell script.

5.5.2 Turning vmod_example into vmod_os

Now that we've been introduced to `vmod_example`, it's time to customize the code, and turn it into your own *VMOD*.

Dependencies

The first thing we need to do is make sure all the dependencies are in place. Just like in the previous part about the *Varnish Software VMOD collection*, we need the following dependencies on *Debian* and *Ubuntu* systems:

```
# apt-get install -y varnish-dev autoconf automake gcc libtool make \
python3-docutils git
```

On *Red Hat*, *Fedora*, and *CentOS* systems, you can use the following command to install the dependencies:

```
# yum install -y varnish-devel autoconf automake gcc libtool make \
python3-docutils git
```

Please note that `git` was also added as a dependency. It is used to retrieve and check out a copy of the repository on your machine.

Downloading these tools allows the `autogen.sh` to generate the software configuration, and eventually generate the `Makefile`.

Getting the code

The best way to get the code is by *cloning the Git repository*, as demonstrated below:

```
git clone https://github.com/varnishcache/libvmod-example.git
```

This creates a `libvmod-example` folder that includes all the code from the repo.

But as explained, we don't really care about `vmod_example`; we want to develop `vmod_os`. That requires some renaming:

```
mv libvmod-example/ libvmod-os/
cd libvmod-os/
./rename-vmod-script os
```

These commands will rename the local folder and will make sure all references to `example` are replaced with `os`.

5.5.3 Looking at the vmod_os.c

Enough with the directory structure and the build scripts: a *VMOD* is all about custom code. Let's take a look at the custom code then.

This is the code we're going to put inside `src/vmod_os.c`:

```
#include "config.h"
#include "cache/cache.h"
#include <sys/utsname.h>
#include "vcc_os_if.h"

VCL_STRING
vmod_uname(VRT_CTX, VCL_BOOL html)
{
    struct utsname uname_data;
    char *uname_str;
    char *br = "";

    CHECK_OBJ_NOTNULL(ctx, VRT_CTX_MAGIC);

    if (uname(&uname_data)) {
        VRT_fail(ctx, "uname() failed");
        return (NULL);
    }

    if (html) {
        br = "<br>";
    }

    uname_str = WS_Printf(ctx->ws,
        "OS: %s%s\n"
        "Release: %s%s\n"
        "Version: %s%s\n"
        "Machine: %s%s\n"
        "Host: %s%s\n",
        uname_data.sysname, br, uname_data.release, br,
        uname_data.version, br, uname_data.machine, br,
        uname_data.nodename, br);

    if (!uname_str) {
        VRT_fail(ctx, "uname() out of workspace");
        return (NULL);
    }

    return (uname_str);
}
```

After having included the header files of our dependencies, we can define the function we want to expose to *VCL*: `uname`.

Conventionally, the *C-function* is then named `vmod_uname()`. When we look at the *function interface*, we notice two arguments:

- `VRT_CTX`, which is a macro that gets replaced by `struct vrt_ctx *ctx` at compile time
- `VCL_BOOL html`, which is an actual argument that will be used in *VCL*

`VRT_CTX` refers to the `vrt_ctx` structure that holds the context of the *VMOD*.

`VCL_BOOL` indicates that the input we receive from *VCL* through this argument will be handled as a boolean with a `true` or `false` value. The name of the argument is `html`.

By enabling this `html` flag, we return the output in HTML format. Otherwise, we just return plain text data.

Within the function, we see the following variable initialization:

```
struct utsname uname_data;
char *uname_str;
char *br = "";
```

- `struct utsname uname_data` is used to initialize the data structure that will hold the `uname` data that will be retrieved from the operating system.
- `char *uname_str` is a string, a *char pointer* to be precise, that will hold the output of the `utsname` structure.
- `char *br = ""` initializes our *line break* variable as an empty string.

`CHECK_OBJ_NOTNULL(ctx, VRT_CTX_MAGIC)` ensures that the context is correctly set before running the `uname()` function. It is part of the consistency self-checks that are present all throughout the core Varnish code, and it helps protect against many nasty forms of bugs.

The following part of the source code calls the `uname()` function and evaluates the output:

```
if (uname(&uname_data)) {
    VRT_fail(ctx, "uname() failed");
    return (NULL);
}
```

`uname()` will store its data in a `utsname` structure. We named this structure `uname_data` and passed it by reference. The output of the function reflects its *exit code*: anything other than zero is an error, according to the documentation for the function, which can be viewed by running the `man 2 uname` command.

If an error does occur, a `VRT_fail()` is executed, which will record this call as a failure in Varnish. As a consequence the whole request will fail, and Varnish will return an *HTTP 503* error to the client that made the request. Varnish will continue to chug along as if nothing happened. It is not a serious problem, just an indication that the VCL execution failed.

The next step is defining the line break format. If the `html` variable is true, the `br` variable will contain `<br>`, which is the HTML equivalent of a line break:

```
if (html) {
    br = "<br>";
}
```

If `html` is `false`, `br` is just an empty string, which is fine for plain text.

Eventually, the output from the `uname_data` structure is extracted, and turned into a string:

```
uname_str = WS_Printf(ctx->ws,
    "OS: %s%s\n"
    "Release: %s%s\n"
    "Version: %s%s\n"
    "Machine: %s%s\n"
    "Host: %s%s\n",
    uname_data.sysname, br, uname_data.release, br,
    uname_data.version, br, uname_data.machine, br,
    uname_data.nodename, br);
```

Using the `WS_Printf()` function, typical `printf()` logic is executed, but *workspace memory* is allocated automatically. The `uname_str` variable where the results are stored can return a string that looks like this:

```
OS: Linux
Release: 4.19.76-linuxkit
Version: #1 SMP Tue May 26 11:42:35 UTC 2020
Machine: x86_64
Host: 19ee8d684eea
```

There is of course no guarantee that the newly formatted string will fit on the workspace, which is quite small on the client side. An extra check is added to the source code to deal with that:

```
if (!uname_str) {
    VRT_fail(ctx, "uname() out of workspace");
    return (NULL);
}
```

If the `uname_str` string is not set, it probably means we ran out of workspace memory. We will once again call `VRT_fail`, as it is an exceptional condition. It would have been possible to just return `NULL` without failing the whole request, but we like things being straight and narrow, reducing the complexity our users have to deal with.

So, finally, when all is good and all checks have passed, we can successfully return the `uname_string`:

```
return (uname_str);
```

This will hand over the string value to the *VCL code* that called `os.uname()` in the first place.

Looking at the vmod_os.vcc

Another crucial file in the `src` folder is `vmod_os.vcc`. This file contains the *API* that is exposed to the *VCL compiler* and is called by the `vmodtool.py` script.

Here's the `vmod_os.vcc` code:

```
$Module os 3 OS VMOD

DESCRIPTION
===========

OS support

$Function STRING uname(BOOL html = 0)

Return the system uname
```

This file defines the name of the module through the `$Module` statement, but it also lists the function API through the `$Function` statement.

The other more verbose information in this file is used to generate the *man pages* for this module.

What we can determine from this file is pretty straightforward:

- There is a *VMOD* called `os`.
- There's a function called `uname()`.
- The `os.uname()` function returns a string.
- The `os.uname()` function takes one argument, which is a boolean called `html`.
- The default value of the `html` argument is *false*.

Building the VMOD

The procedure that is used to build `vmod_os` is quite similar to the procedure we used to build the *Varnish Software VMOD collection*. Only the first shell script to initialize the software configuration is different.

Be sure to install build dependencies like `autoconf`, `automake`, `make`, `gcc`, `libtool`, and `rst2man` before proceeding.

Here's what you have to run:

```
./autogen.sh
./configure
make
make install
```

- `./autogen.sh` will prepare the software configuration file using tools like `auto-conf` and `automake`.
- `./configure` is a shell script that is generated in the previous step and that prepares the build configuration. It also generates the `Makefile`.
- `make` will actually compile the source code using the `gcc` compiler.
- `make install` will use `libtool` to bundle the compiled files into an `.so` file and will put the library file in the right directory.

Testing the VMOD

Once the *VMOD* is built, we can test whether it behaves as we would expect. In the `src/tests` folder, you can edit `test01.vtc` and put in the following content:

```
varnishtest "Test os vmod"

server s1 {
} -start

varnish v1 -vcl+backend {
    import ${vmod_os};

    sub vcl_recv {
        return (synth(200, "UNAME"));
    }

    sub vcl_synth {
        synthetic(os.uname());
        return (deliver);
    }
} -start

client c1 {
    txreq
    rxresp
    expect resp.body ~ "OS:"
} -run
```

This `vtc` file contains the syntax that is required to perform functional tests in *Varnish*. This test case has three components:

- A `server` that acts as the origin
- A `varnish` that processed the *VCL*
- A `client` that sends requests to `varnish`

In this case, the `server` does absolutely nothing, because `varnish` will return synthetic output based on the *VMOD* that it imports. The `${vmo_os}` statement will dynamically parse in the location of the built *VMOD*. The syntax is composed such that it can be executed before `make install` is called.

You can run the tests by calling `make check`. This can be done right after `make` and before `make install` if you want.

This is the output that you get:

```
PASS: tests/test01.vtc
======================================================================
=======
Testsuite summary for libvmod-os 0.1
======================================================================
=======
# TOTAL: 1
# PASS:  1
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
======================================================================
=======
```

All is good, and the test has passed. The assertion from the test is that the output from the `os.uname()` call will contain the `OS:` string.

`test01.vtc` can also be called via `varnishtest`, but it requires `${vmod_os}` to be replaced with `os`.

The following command can then be run:

```
varnishtest src/tests/test01.vtc
```

And the output will be the following:

```
#    top  TEST test01.vtc passed (5.221)
```

Using the VMOD

Once the build is completed, and `make install` is executed, `libvmod_os.so` will be stored in the path that corresponds to the `vmod_path` runtime parameter.

Then you can safely import the *VMOD* into your *VCL file* using `import os;`. Here's a *VCL file* that uses are custom *VMOD*:

```
vcl 4.1;

import os;

backend default none;

sub vcl_recv
{
    if (req.url == "/uname") {
        return (synth(200, "UNAME"));
    }
}

sub vcl_synth
{
    if (resp.reason == "UNAME") {
        if (req.http.User-Agent ~ "curl") {
            synthetic(os.uname());
        } else {
            set resp.http.Content-Type = "text/html";
            synthetic(os.uname(html = true));
        }
        return (deliver);
    }
}
```

The output when `/uname` is called, could be the following:

```
OS: Linux
Release: 4.19.76-linuxkit
Version: #1 SMP Tue May 26 11:42:35 UTC 2020
Machine: x86_64
Host: 19ee8d684eea
```

This *VCL example* will use *plain text* output when the `curl` *User-Agent* is used. For any other browser, a `<br>` line break will be added to each line, and the `text/html` *Content-Type* header is added.

5.6 Summary

VCL is a powerful and flexible cache configuration language, but *out of the box* it only offers you the tools to handle *requests*, *responses*, *objects*, *session information*, and *VCL state transitions*.

For all other things, you rely on *VMODs*. Your *Varnish* installation comes with a set of *in-tree VMODs* that were purposely kept outside of the core code.

All the cool integrations are done using *VMODs*; everything related to *decision-making on the edge* uses *VMODs*.

Now you certainly realize that there is a *VMOD ecosystem* that has various types of stakeholders in it: the *Varnish Cache* team, *Varnish Software's* development team, and even individuals in the community.

Getting access to those modules shouldn't be a frightening endeavor at all: sometimes there are distro packages, certain modules are *in-tree* and shipped by default. If you use *Varnish Enterprise*, all the enterprise *VMODs* and a collection of open source *VMODs* are already installed as part of the product.

And if none of these things apply to you: compiling *VMODs* from source is pretty straightforward. There will usually be line-by-line instructions on how to build and install the *VMOD* on your computer.

Hopefully this chapter was an eye opener. But now it's time to turn the page, and go to the next chapter. In *chapter 6*, we'll cover *cache invalidation*, which is all about removing objects from cache. We'll even use some *VMODs* for that.

Chapter 6: Invalidating the cache

Welcome to *chapter 6*. Let's kick this one off with a very powerful saying:

There's only one thing worse than not caching enough, and that is caching for too long.

Allow us to elaborate.

The only detrimental effect of uncacheable content, or content with a low *TTL*, is a low hit rate. By now you should appreciate the direct impact of a good hit rate. Although latency or downtime are potential consequences of a poor hit rate, the alternative could be much worse.

If you cache for too long, the cached content might no longer represent the actual state of the origin content. Data inconsistency is the main effect, and in a lot of situations, that's really bad.

Imagine the news industry: online newspapers heavily depend on caches to cope with traffic spikes. But when they have a scoop, the breaking news item should immediately be visible on their homepage.

But when a cache like *Varnish* is used, as long as the *TTL* hasn't expired, the breaking news item is nowhere to be found.

The same applies to live video content delivery: the manifest files, which contain references to the video files, may change depending on what the camera records. For live events it is very important to have consistent content.

It is commonly believed that it's not worth caching the manifest files because they often change, but in most cases it is not entirely true: manifest files can be cached for a short period of time as long as you have a cache invalidation plan to not evict outdated manifest files from the cache.

What's the solution to this problem? Lowering your *TTLs*? No, not really, because it's not a dilemma of having to choose between a high hit rate and data consistency.

There is a door number three, which is *invalidating cached objects*.

Varnish has various mechanisms to mark objects as expired or to forcefully remove objects from cache. Endpoints and interfaces are offered for external applications to trigger these invalidations.

When you look at plugins and integrations from popular open source content management and e-commerce frameworks like *Drupal*, *WordPress*, *Magento*, *Joomla* and *Prestashop*, you'll see that their main focus is hooking into their content management logic and integrating endpoints to *Varnish*'s cache invalidation.

This chapter will focus on these mechanisms to remove objects from the cache. The main objectives being data consistency and freeing up space in the cache.

6.1 Purging

The most basic and easy-to-use *cache invalidation mechanism* in Varnish is without a doubt *purging*.

The idea behind purging is that you can perform a `return(purge)` in `vcl_recv`, and *Varnish* will remove the object. This would free up space in the cache, either in memory or on disk, after an object lookup.

This means that the *hash* of the object is used to identify it, but `return(purge)` would remove it along with its variants. This isn't *out-of-the-box* behavior; you won't find it in the *built-in VCL*. You need to write some logic for it.

6.1.1 Purge VCL code

The code you need to perform a *purge* is very straightforward but does require some safety measures to be put in place:

```
vcl 4.1;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405, "Not Allowed"));
        }
        return (purge);
    }
}
```

As you can see, the *VCL code* starts with an *ACL definition*. This is very important because you don't want to expose your *purging endpoint* to the public. People could get very creative with this, and it could potentially tank your hit rate.

There's an *if-statement* in place that checks the *ACL* and returns an `HTTP 405 Method Not Allowed` error for unauthorized traffic.

The fact that we choose an *HTTP 405* status means we're not using a regular HTTP `GET` method. Instead we're using a custom `PURGE` method.

Please note that the request method used for purging could be any of the official HTTP request methods or it could be, as in this case, a custom method.

If the *purger* calls the *URL* using a `PURGE` method, and the *ACL* allows the client, we can do a `return(purge)`;

> Be sure to perform some kind of `return()` call, otherwise the *built-in VCL* will kick in as a fallback, and will perform a `return(pipe)` because it doesn't recognize the request method.

6.1.2 Triggering a purge

```
PURGE / HTTP/1.1
Host: example.com
```

The response you get might look like this:

```
HTTP/1.1 200 Purged
Date: Tue, 20 Oct 2020 13:30:12 GMT
Server: Varnish
X-Varnish: 32770
Content-Type: text/html; charset=utf-8
Retry-After: 5
Content-Length: 240
Accept-Ranges: bytes
Connection: keep-alive

<!DOCTYPE html>
<html>
  <head>
    <title>200 Purged</title>
  </head>
  <body>
    <h1>Error 200 Purged</h1>
    <p>Purged</p>
    <h3>Guru Meditation:</h3>
    <p>XID: 32770</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html
```

As you can see, `return(purge)` uses the `vcl_synth` subroutine to return a *synthetic HTTP response*.

But since `return(purge)` does a *cache lookup* first, it uses the *host header* and *URL* to identify the object. This means purging happens on a *per-URL and a per-host basis*. Note that when purging the same exact parameters used to define the *hash key* must be reported by the purge command. The standard parameters are the *hostname* and the *URI*.

If you want to purge the `/contact` page, you need call the right *URL* in your purge request:

```
PURGE /contact HTTP/1.1
Host: example.com
```

If your *Varnish setup* allows cached content for the `foo.com` hostname, the appropriate `Host` header needs to be added to the purge call:

```
PURGE /contact HTTP/1.1
Host: foo.com
```

In case you wonder: yes, purges can also happen over `HTTP/2` if the feature was enabled via the `-p feature=+http2` runtime parameter.

6.1.3 vmod_purge

There's also a `vmod_purge` that is shipped with *Varnish 6* that offers slightly more features.

The *VMOD* offers a `purge.hard()` and a `purge.soft()`. The *hard purge* will immediately expire the object, whereas the *soft purge* will re-arm the object with custom settings for *TTL*, *grace*, and *keep* values.

Hard purge

Here's an example where `purge.hard()` is used:

```
vcl 4.1;

import purge;

acl purge_acl {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (client.ip !~ purge_acl) {
            return (synth(405));
        }
        return (hash);
    }
}

sub my_purge {
    set req.http.purged = purge.hard();
    if (req.http.purged == "0") {
        return (synth(404));
    }
    else {
        return (synth(200, req.http.purged + " items purged."));
    }
}

sub vcl_hit {
    if (req.method == "PURGE") {
        call my_purge;
    }
}

sub vcl_miss {
    if (req.method == "PURGE") {
        call my_purge;
    }
}
```

Because we're not performing a `return(purge)`, we actually rely on checks in `vcl_hit` and `vcl_miss`. The extra functionality we're getting is knowing whether or not the *purge call* matched any objects.

If no objects were matched, an `HTTP 404 Not Found` status code is returned. Otherwise the number of matched objects is added to the output.

Soft purge

A *soft purge* will re-arm the object with new *TTL*, *grace*, and *keep* values.

The reality is that `purge.hard()` does this too, as well as `return(purge)`. The only difference is that *regular purges* and *hard purges* set the *TTL* to zero, whereas *soft purges* give you the opportunity to customize those values.

Purging, regardless of the type, will make sure an object is expired, either immediately, or if using soft purge, after the set TTL. Once expired, the *expiry thread* will eventually remove the object from the cache.

If you call `purge.soft(0s, 0s, 0s)`, you'll cause the same effect as `purge.hard()` or `return(purge)`.

The benefit is the flexibility you get, which is illustrated in the example below:

```
vcl 4.1;

import purge;
import std;

acl purge_acl {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (client.ip !~ purge_acl) {
            return (synth(405));
        }
        return (hash);
    }
}

sub my_purge {
    set req.http.purged = purge.soft(std.duration(req.http.ttl,0s),
        std.duration(req.http.grace,0s),
        std.duration(req.http.keep,0s));

    if (req.http.purged == "0") {
        return (synth(404));
    }
    else {
        return (synth(200, req.http.purged + " items purged."));
    }
}
```

```
sub vcl_hit {
    if (req.method == "PURGE") {
        call my_purge;
    }
}

sub vcl_miss {
    if (req.method == "PURGE") {
        call my_purge;
    }
}
```

This example allows you to set the *TTL* through a custom `ttl` request header. The same applies to *grace* and *keep* via the respective `grace` and `keep` request headers.

If we want to *soft purge* the homepage, but we want to make sure there's *one minute of grace* left, you'll perform the following *HTTP request*:

```
PURGE /contact HTTP/1.1
Host: example.com
Ttl: 0s
Grace: 60s
```

We didn't specify the `keep` header, but the *VCL* uses `0` as the fallback value for each of these headers.

6.1.4 Integrating purge calls in your application

It is easy to perform a *purge* on the command line using `curl` or `HTTPie`, as illustrated below:

```
#HTTPie
http PURGE "www.example.com/foo"
# curl
curl -X PURGE "www.example.com/foo"
```

In reality, you'll probably use the *HTTP client library* that comes with your application framework.

For frameworks like *WordPress*, *Drupal*, *Magento*, and many others, there are community-maintained plugins available that perform *purge calls* to *Varnish*. The *VCL* to handle the purges is also included.

Consider this a segue to the next segment, as a lot of applications cannot rely only on *purging* because it is too limited. If you change content that affects many *URLs*, you'll have to perform a lot of *purge calls*, which might not be very efficient.

A lot of these *framework plugins* will use *bans* instead, or combine *purges and bans*. Let's go to the next section to talk about these so-called *bans* and why they are so popular.

6.2 Banning

Banning is a concept in *Varnish* that allows *expression-based cache invalidation*. This means that you can invalidate multiple objects from the cache without the need for individual *purge calls*.

A *ban* is created by adding a *ban expression* to the *ban list*. All objects in the cache will be evaluated against the expressions in the *ban list* before being served. If the object is banned Varnish will mark it as expired and fetch new content from the backend.

6.2.1 Ban expressions

A *ban expression* consists of *fields*, *operators*, and *arguments*. Expressions can be chained using the `&&` operator. Only *logical AND* operations can be performed. *Logical OR* operations are done by evaluating multiple ban expressions.

Expression format

This is the format of *ban expressions*:

```
<field> <operator> <arg> [&& <field> <oper> <arg> ...]
```

The following *fields* are supported:

- `req.url`: the request URL
- `req.http.*`: any request header
- `obj.status`: the cache object status
- `obj.http.*`: the response headers stored in the cached object

These fields look familiar, and they represent some of the objects and variables in *VCL*.

The *operator* can be:

- `==`: the `field` equals an `arg`
- `!=`: the `field` is not equal to an `arg`
- `~`: the `field` matches a regular expression defined by the `arg`
- `!~`: the `field` doesn't match a regular expression defined by the `arg`

The *argument* of a *ban expression* is either a literal string or a regular expression pattern. Strings are not delimited by double quotes `"` or the long string format `{"..."}`.

Expression examples

Let's start with a very basic example that is the *ban* equivalent of a regular *purge*:

```
req.url == / && req.http.host == example.com
```

So the request's *URL* equals `/`, and the request's `Host` header equals `example.com`.

In another example we'll invalidate all objects from the cache that have an *HTTP 404* status:

```
obj.status == 404
```

We can also create an expression that uses *response headers* that are stored in the object. Let's say we want to invalidate all images at once. We'd use the following expression:

```
obj.http.Content-Type ~ ^image/
```

This expression looks at the `Content-Type` response header and invalidates all items that match the `^image/` regular expression.

For the last example, we'll match on a *URL pattern*, instead of an individual *URL* :

```
req.url ~^/products(/.+|$) && req.http.host == example.com
```

This pattern will match all objects where the URL starts with `/products/...` or equals `/products`.

6.2.2 Executing a ban from the command line

Now that you know what a *ban* is and what *ban expressions* look like, it's time to explain how to execute such a ban.

The quickest way to do this is by using the `varnishadm` program. This program makes a connection to the *CLI interface* of `varnishd`.

You can choose to call the `varnishadm` program without any arguments, where you can enter individual commands. This is what happens in the example below:

```
$ varnishadm
200
-----------------------------
Varnish Cache CLI 1.0
-----------------------------

Type 'help' for command list.
Type 'quit' to close CLI session.

varnish> ban obj.status == 404
200
```

The `ban obj.status == 404` command will issue a *ban* that aims to invalidate all objects with an *HTTP 404* status code.

Another way you can ban using `varnishadm` is by adding the ban expression as an argument. Here's an example of this:

```
varnishadm ban obj.status == 404
```

Sometimes certain characters in your *ban expression* might interfere with how your *Linux shell* interprets commands:

```
$ varnishadm ban obj.http.Content-Type ~ ^image/
expected conditional (~, !~, == or !=) got "/root"
Command failed with error code 106
```

In that case, you're better off using quotes to avoid errors:

```
varnishadm ban "obj.http.Content-Type ~ ^image/"
```

6.2.3 Ban VCL code

Although banning can be done using `varnishadm` and doesn't require a *VCL* implementation, it would be nice to use the `BAN` method to invalidate objects via *bans*. Much like our *purging* example.

We could have exactly the same behavior, but we strip out the *purge* logic from under the hood, and replace it with *ban* logic. *VCL* provides a `ban()` function that takes the *ban expression* as the argument.

Purge replacement

The following example is an exact copy of the *purge VCL example*, but instead of performing a `return(purge)`, we return a synthetic response and use the `ban()` function to execute the *ban*:

```
vcl 4.1;

acl banners {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "BAN") {
        if (!client.ip ~ banners) {
            return(synth(405));
        }
        ban("req.url == " + req.url
            + " && req.http.host == " + req.http.host);
        return(synth(200,"Ban added"));
    }
}
```

We have now created a *purge* replacement, but didn't gain any flexibility.

Invalidate URL patterns

A more flexible approach would be to invalidate *URL patterns* rather than individual *URLs*. We could send a custom request header that contains this pattern.

The following example will enforce the custom `x-ban-pattern` request header to be set:

```
vcl 4.1;

acl banners {
    "localhost";
    "192.168.55.0"/24;
}
```

```
sub vcl_recv {
    if (req.method == "BAN") {
        if (!client.ip ~ banners) {
            return(synth(405));
        }
        if(!req.http.x-ban-pattern) {
            return(synth(400,"Missing x-ban-pattern header"));
        }
        ban("req.url ~ " + req.http.x-ban-pattern
            + " && req.http.host == " + req.http.host);
        return (synth(200,"Ban added"));
    }
}
```

This ban would be triggered using the following *HTTP request*:

```
BAN / HTTP/1.1
Host: example.com
X-Ban-Pattern: ^/products/[0-9]+
```

The ban we just issued using *HTTP*, will result in the following ban expression:

```
req.url ~ ^/products/[0-9]+ && req.http.host == example.com
```

Long story short: we are banning all objects where the *URL* starts with `/products/`, followed by a *numeric value*. It is an open-ended regular expression, so *URLs* containing even more data after the *numeric value* will also match.

Complete flexibility

We can even take it up a notch, and allow even more flexibility, to the extent that the user is responsible for formulating the complete *ban expression*.

Here's such an example:

```
vcl 4.1;

acl banners {
    "localhost";
    "192.168.55.0"/24;
}
```

```
sub vcl_recv {
    if (req.method == "BAN") {
        if (!client.ip ~ banners) {
            return(synth(405));
        }
        if(!req.http.x-ban-expression) {
            return(synth(400,"Missing x-ban-expression header"));
        }
        ban(req.http.x-ban-expression);
        return (synth(200,"Ban added"));
    }
}
```

This *VCL code* would then be invoked using the following *HTTP request*:

```
BAN / HTTP/1.1
Host: example.com
X-Ban-Expression: obj.status == 404
```

The advantage here is that you're not restricted to the *request context*, and you can also match other fields.

The downside is that you're exposing the complexity of *ban expressions* to the end user. Instead, the concept of *URL patterns* would seem more intuitive for users.

The best of both worlds

Sometimes you don't want to choose and just want to have it all:

- Regular *purges* when you want to invalidate an individual *URL*
- *Bans* when you want to invalidate a *URL pattern*

This can be done with a single implementation. Whereas we returned an *HTTP 400* status when the `x-invalidate-pattern` header was not set, we can use *purging* as a fallback instead.

Here's the code:

```
vcl 4.1;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-invalidate-pattern) {
            return(purge);
        }
        ban("req.url ~ " + req.http.x-invalidate-pattern
            + " && req.http.host == " + req.http.host);
        return (synth(200,"Ban added"));
    }
}
```

So if you just want to purge the `/products` page, you can issue the following *HTTP request*:

```
PURGE /products HTTP/1.1
Host: example.com
```

But if you want to invalidate all subordinate resources of `/products/`, you can add the `x-invalidate-pattern` header and specify the *URL pattern*:

```
PURGE / HTTP/1.1
Host: example.com
X-Invalidate-Pattern: ^/products/
```

6.2.4 The ban list

Unlike *purges*, *bans* will not immediately remove objects from the cache. The synthetic message from the *VCL examples* already gave it away: *bans are added*.

When you execute a *ban*, the *ban expression* is added to the *ban list*. This is a list containing all the bans that need to be evaluated, and matched against all the objects in cache.

The easiest way to see the contents of the *ban list* is by running `varnishadm ban.list`.

There is always an item on the list

Here's the output of the *ban list* when the `varnishd` process was just started:

```
$ varnishadm ban.list
Present bans:
1603270370.244746     0 C
```

Although no bans were issued, and no objects are stored in the cache, there is already an item on the list. Let's break it down:

- `1603270370.244746` is the time at which the ban was added. It is in *Unix timestamp* format and has microsecond precision.
- `0` is the refcount. There are currently `0` objects that refer to this ban.
- `C` stands for *Completed*. This means the *ban* is fully evaluated.

The reason there is already a *ban* on the list is because every object in cache needs to refer to the last *ban* it has seen when entering the cache. This allows *bans* that are older than the object to be disregarded.

So as soon as the first object is stored in cache, the *refcount* increases to `1`:

```
$ varnishadm ban.list
Present bans:
1603270370.244746     1 C
```

The *refcount* will increase as objects get inserted.

Adding a first ban

The *ban list* will change as soon as the first *ban* is added.

The following example looks a bit weird:

```
varnishadm ban obj.status != 0
```

We're banning all objects that do not have a `0` status. That's literally every object in the cache.

When we consult the *ban list*, we see it has been added:

```
$ varnishadm ban.list
Present bans:
1603272627.622051     0 -  obj.status != 0
1603270370.244746     3 C
```

Initially all three objects still refer to the initial *ban* as the one they have seen last. But with the addition of the new ban, that will change.

After a short while, the *ban list* will look like this:

```
$ varnishadm ban.list
Present bans:
1603272627.622051     0 C
1603270370.244746     0 C
```

The newly added *ban* is completed, and no objects refer to it because we just removed all objects from the cache. The initial *ban* is also still around.

As soon as a new object enters the cache, it refers to the last one it has seen:

```
$ varnishadm ban.list
Present bans:
1603272627.622051     1 C
```

If you look at the timestamp, it is `1603272627.622051`, which matches the *ban* we just executed.

Adding multiple bans

Let's have a look at a *ban list* that already has some *ban expressions* on it:

```
$ varnishadm ban.list
Present bans:
1603273224.960953     2 -  req.url ~ ^/[a-z]$
1603273216.857785     0 -  req.url ~ ^/[a-z]+/[0-9]+
1603272627.622051     9 C
```

Nine objects saw `1603272627.622051` as their last ban. This means that up to two *ban expressions* should be evaluated for those objects.

For two objects, `1603273224.960953` was the last one they saw. These objects aren't subject to any invalidation. These were objects that were inserted into to cache after the two recent *bans* were added.

There are zero objects that saw `1603273216.857785` as their last *ban*. This kind of makes sense because if you do the math between the last and the second-to-last ban, you'll see there's only an eight second difference between the two bans. During those eight seconds, no new objects got added to the cache.

As time progresses, you'll see that the `req.url ~ ^/[a-z]+/[0-9]+` evaluation has completed, and that those nine objects have been processed:

```
$ varnishadm ban.list
Present bans:
1603273224.960953     2 -   req.url ~ ^/[a-z]$
```

This means that nine objects were invalidated since they are no longer referenced.

Any future *bans* that are executed will apply to the two remaining objects, as long as they have not expired.

6.2.5 The ban lurker

We have to be honest: there is one piece of crucial information we held back from you.

Throughout this section about *banning*, we talked about *ban expressions*, the *ban list*, and about objects being matched. But we never mentioned what mechanism is responsible for that.

There is a thread, which was mentioned in the *Under the hood* section of *chapter 1*, that is called the *ban lurker*.

This thread will inspect the *ban list*, and match the *ban expression* to the right objects.

Runtime parameters

The *ban lurker* thread has some *runtime parameters* that control its behavior:

- `ban_lurker_age` is the minimum age an object should have before it is processed by the *ban lurker*. The default value is `60` seconds.
- `ban_lurker_sleep` is the number of seconds the *ban lurker* sleeps before processing another batch. The default value is `0.010` seconds.

- `ban_lurker_batch` is the number of bans the *ban lurker* processed before going back to sleep. The default value is `1000`.
- `ban_lurker_holdoff` sets the number of seconds the *ban lurker* holds off when lock contention occurs during a cache lookup. The default value is `0.010` seconds.
- `ban_cutoff` limits the *ban lurker* from inspecting the ban list until the *ban_cutoff* limit is reached; beyond that it treats all objects as if they matched a ban and removes them from cache. The default value is `0`.
- `ban_dup` eliminates older identical bans when a new ban is added. The default value is `on`.

Ban lurker workflow

Every *0.010 seconds* the *ban lurker* will look for objects that are at least *one minute old*. The lurker will process *1000 at a time*. It looks for the position of that object on the *ban list* and applies the most recent bans up until the point when a *ban expression* matches.

When a match is found that object is put on the expiry list and is removed from the cache shortly thereafter.

Ban lurker scope

Because the *ban lurker* is a separate thread that has no knowledge of any incoming *HTTP request*, its scope is limited to the *object context*.

Any *ban expression* that refers to an `obj.http.*` or an `obj.status` field can be processed by the *ban lurker*. Basically only the *response information* that is part of the object is available to the *ban lurker*.

This begs the question: how do expressions that contain `req.url` or `req.http.*` get processed? It's obvious that these *bans* are not the responsibility of the *ban lurker*.

When the *request context* is used in a *ban expression*, the *worker thread* that handles the incoming request is responsible for this.

This means that such *bans* aren't processed asynchronously and that space is only freed from the cache when a request comes in that matches one of these *ban expressions*.

We'll talk about the performance impact of synchronous bans in one of the next sections.

6.2.6 Enforcing asynchronous bans

Now that we know the scope of the *ban lurker*, and the fact that the *worker thread* is responsible for handling *bans* within the *request scope*, it seems as though *request-based ban expressions* cannot be used in an efficient way.

To that we say:

> Use your imagination, and be creative.

If this were a Tweet or a Facebook post, we would have added an emoji.

If the object has no information about the request, add this information in *VCL*:

```
sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
    set beresp.http.x-host = bereq.http.host;
}
```

These two custom headers basically store the *request context* as custom response headers.

And then you can trust that the *ban lurker* will be able to evaluate the following expression:

```
obj.http.x-url == / && obj.http.x-host == example.com
```

This is what we call *lurker-friendly bans*. As this is quite the mouthful, we can also just call them *asynchronous bans*.

Let's take our *best-of-both-worlds* example, and apply *asynchronous banning*:

```
vcl 4.1;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-invalidate-pattern) {
                return(purge);
            }
            ban("obj.http.x-url ~ " + req.http.x-invalidate-pattern
                + " && obj.http.x-host == " + req.http.host);
            return (synth(200,"Ban added"));
        }
}

sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
    set beresp.http.x-host = bereq.http.host;
}

sub vcl_deliver {
    unset resp.http.x-url;
    unset resp.http.x-host;
}
```

To make this work, we had to add the `beresp.http.x-url` and `beresp.http.x-host` *URLs*, but we also have to clean them up upon delivery. And the fields we're matching in the `ban()` function now reflect these two custom response headers.

And then you can send the following *HTTP request* to *Varnish*:

```
PURGE / HTTP/1.1
Host: example.com
X-Invalidate-Pattern: ^/products/
```

After the object reaches the `ban_lurker_age` value, the *ban lurker* will come in and expire the object. Unlike *synchronous bans* within the request scope, the *worker thread* won't have to do the heavy lifting, the *ban lurker* will.

6.2.7 Tag-based invalidation

Most *cache invalidation* implementations focus solely on the *URL* as a way to identify and invalidate objects. This only works if the content in your application can easily be mapped to one or more *URLs*.

But for more advanced content that appears all over the place, it is nearly impossible to map this to the right *URLs*.

An alternative approach is to *tag* content, and *ban* objects based on these tags.

Consider the following *HTTP response*:

```
HTTP/1.1 200 OK
Cache-Control: public, s-maxage=60
Tags: tag1 tag2 tag3
...
```

This response will be stored in cache for a minute, but if you want to get the corresponding object out of cache earlier, you can issue a ban that matches a *tag* from the `Tags` header.

The following *ban expression* would remove every object from cache that has `tag1` in its `Tags` header:

```
obj.http.tags ~ tag1
```

You can also include this in your *VCL code*:

```
vcl 4.1;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-ban-tag) {
            return(synth(400,"x-ban-tag missing"));
        }
        ban("obj.http.tags ~ " + req.http.x-ban-tag);
        return (synth(200,"Ban added"));
    }
}
```

Triggering the *ban* via *HTTP* could result in the following *HTTP request*:

```
PURGE / HTTP/1.1
Host: example.com
X-Ban-Tag: tag1
```

6.2.8 Integrating bans in your application

Just like *purges*, you can call *bans* using command line *HTTP clients*:

```
#HTTPie
http PURGE "www.example.com" "X-Purge-Pattern:^/contact"
# curl
curl -X PURGE -H "X-Purge-Pattern:^/contact" "www.example.com"
```

But as we've seen earlier, there are other command line tools in place to trigger *bans*:

```
varnishadm ban obj.http.Content-Type ~ ^image/
```

And for WordPress, Drupal, Magento, and many others, there are community-maintained plugins available that perform *bans* in *Varnish*.

But not all of these frameworks implement their *cache purging logic* using an *HTTP endpoint*. Magento, for example, connects to the *Varnish command line* over a *TCP socket*.

We'll talk about the *Varnish CLI socket protocol* in *chapter 7*.

6.2.9 Ban limitations

If you factor in the scope of *bans*, and enforce *request-based bans* to be *lurker friendly*, it does seem like a great solution. For most people it is.

However, *banning* is architected in such a way that it can become a very CPU-intensive process.

Because every object `(n)` has to be matched against all remaining *ban expressions* `(m)`, the complexity is `n * m`. This means that if you have a lot of objects and a lot of *bans*, a lot of computations need to happen.

For *asynchronous* bans, the burden is on the *ban lurker* thread. But for *synchronous bans*, the *worker thread* is responsible for processing request-related items on the ban list. In that case, the computationally heavy logic might slow down the request.

Potential performance issues related to *bans* also depend on the quality of the regular expression that is used: the more complicated the *regex*, the longer it takes to process.

Header matching will also have an impact because the *ban lurker* or the *worker thread* needs to cycle through all headers until the matching header is found.

This adds extra complexity, and the more headers a request or response has, the more work that needs to happen.

Long story short: the unparalleled flexibility of *bans* comes at a cost. Perhaps like all things in life.

6.3 Secondary keys

Referring to a point we've already made a couple of times:

> Most cache invalidation strategies are based on the *URL* of a request. This only works if the content in your application can easily be mapped to one or more *URLs*.

Sometimes a content change impacts many *URLs*, and sometimes it is impossible to know which *URLs* will need to be evaluated. Under those circumstances, *banning and purging* doesn't work.

We already hinted at *tag-based invalidation* in the previous section of the book.

Instead of identifying objects in the cache based on their *request URL*, you can use arbitrary tags to identify objects. By invalidating this tag, all objects are purged from the cache at once.

For *request-based invalidation*, we go through the typical *lookup logic* that is triggered in the `vcl_hash` subroutine: we take the *URL* and the `Host` header, and turn this into a *hash key*. This key can be considered the *primary key*.

But if we start using other identifiers to match objects, such as tags, we can say that there's a *secondary key* involved. Hence, the name of the section.

Although the `ban("obj.http.tags ~ " + req.http.x-ban-tag)` example that we saw earlier works, it is not really built for the job.

Varnish has two *VMODs* that store secondary keys for objects, which allow these objects to be purged based on these *secondary keys*:

- `vmod_xkey` is an open source *VMOD* that is part of the *Varnish Software VMOD collection*.
- `vmod_ykey` is the successor of `vmod_xkey`. It is only available in *Varnish Enterprise*.

Let's talk about those *VMODS* for a minute.

6.3.1 vmod_xkey

`vmod_xkey` is part of the *Varnish Software VMOD collection*. It is open source, and its *API* can be found at https://github.com/varnish/varnish-modules/blob/master/src/

vmod_xkey.vcc.

The *API* for this *VMOD* is pretty simple. There are only two functions:

- `xkey.purge()`
- `xkey.softpurge()`

Both functions take a string as an argument. This string refers to the key that needs to be purged. This string may contain an individual key or a space-separated list of keys.

Initializing vmod_xkey

The initialization of `vmod_xkey` happens automatically. As soon as `import xkey;` is part of your *VCL* file, `vmod_xkey` will be bootstrapped, and any new objects that are inserted in cache will be analyzed.

Xkey will look for the `xkey` or the `X-HashTwo` response headers and will register the tags that are exposed through these headers.

Registering keys

As mentioned, `vmod_xkey` will look for the `xkey` or the `X-HashTwo` response headers. Xkey headers are normally added by the backend application, but you can also add them in the `vcl_backend_response` subroutine. Multiple keys in one header line are separated by spaces or commas.

So if you want the keys `category_sports`, `id_1265778`, `type_article` for a page on a news website, the response would look like this:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Cache-Control: public, s-maxage=60
Xkey: category_sports id_1265778 type_article
```

Invalidating content

Once `vmod_xkey` is imported, and `xkey` headers of objects are processed, we have a collection of *secondary keys* that can be used for invalidation.

If for some reason all articles from the *sports category* need to be purged from cache, it's just a matter of purging the `category_sports` key.

For the implementation of the `vmod_xkey` invalidation logic, we can revisit the *tag-based invalidation example* that used *bans*. It's just a matter of swapping out the `ban()` function with the corresponding `xkey.purge()` function, and some cosmetic changes:

```
vcl 4.1;

import xkey;
import std;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-xkey-purge) {
            return(synth(400,"x-xkey-purge header missing"));
        }

        set req.http.x-purges = xkey.purge(req.http.x-xkey-purge);

        if (std.integer(req.http.x-purges,0) != 0) {
             return(synth(200, req.http.x-purges + " objects
purged"));
        } else {
            return(synth(404, "Key not found"));
        }
    }
}
```

If we look back at the example of our news website, we can use the following *HTTP request* to invalidate all news articles from the *sports category*:

```
PURGE / HTTP/1.1
Host: example.com
X-Xkey-Purge: category_sports
```

If no matching keys were found, you'll get an *HTTP 404* response; otherwise you'll get a regular *HTTP 200* response containing the number of purged objects.

vmod_xkey limitations

We started this chapter talking about *purging*. It's simple, it's effective, but it's not really flexible. Then we introduced *banning*, which seemed like the perfect alternative, but the flexibility comes a cost.

So here we are, talking about `vmod_xkey` as a more powerful alternative for *tag-based invalidation*. The cost of invalidating many objects with many keys is a lot lower than for *bans*.

But there is still a cost, some limitations, and some unpleasant side effects.

`vmod_xkey` doesn't scale that well because of its architecture. It is not a core concept, but rather an afterthought that was introduced in the form of a *VMOD*. The *Varnish core* doesn't have a framework in place to natively support *secondary keys* alongside other parts of the core.

Locking

This means *Xkey* had to look for an existing mechanism in *Varnish* that allowed it to safely invalidate content in a *multi-threaded context*.

The *dynamic data structure* that is responsible for object expiry seemed like a good match. The `vmod_xkey`s interaction with *Varnish* is basically a bit forced and piggy-backs on the expiry mechanism for safe access to objects.

The cost that *Xkey* incurs, both during object insertion and eviction, is added to the expiry data structure. This is mainly due to locking.

While `vmod_xkey` processes keys during object insertions, or purges objects using keys, it uses the *mutexes* of the *expiry data structure* for locking. This ensures safe access to these objects, but also blocks anything else from accessing the expiry mechanism.

This can really bring *Varnish* to its knees on busy sites where many new objects are inserted or a lot of purges happen.

Old objects aren't processed

Another limitation is that `vmod_xkey` only processes newly inserted objects. Objects that were already in the cache before `import xkey;` took place cannot be purged.

This limitation only occurs when `varnishd` was started using a *VCL file* that did not import `vmod_xkey`. Any object that was inserted using that *VCL configuration* will not be subject to *secondary key* inspection.

This can become a tangible issue in cases where *custom VCL* is deployed to your *Varnish server* as part of a config management strategy: your *Varnish* server is first started using *boilerplate VCL*, and at a later stage in the setup, the config management system deploys a *VCL config* that uses *Xkey*.

Performs poorly with persisted MSE caches

A third known limitation of *Xkey* is the fact that it behaves very poorly on *persisted MSE caches*.

The *Massive Storage Engine* supports *cache persistence* by storing objects on disk, while *hot objects* are kept in memory. Although this is a highly optimized storage component, *Xkey* doesn't manage to benefit from this.

Imagine having a persisted cache with one million objects. When `vmod_xkey` is imported by the *VCL*, the persisted cache will look like new objects to *Xkey*, and it will start analyzing them one by one.

This analysis process consists of inspecting the `Xkey` response header. This means cycling to the header object that is stored in disk. If you have one million persisted objects stored in cache, *one million disk operations* need to take place.

The secondary keys that result from the lookups in cache also need to be composed, which is *CPU-intensive*. Although this pales in comparison to the *disk I/O* that is caused by *Xkey*'s object indexing.

This can make the startup of *Varnish* extremely slow.

The locking effect that was previously discussed will only be amplified by the use of persisted MSE caches: the waits will be longer, the locking will last longer, and it will take more time for resources to be freed.

It is important to understand that not every *Varnish* setup will suffer from these performance issues. It's a matter of scale: the number of requests your *Varnish* server receives, the number of objects in cache, the number of purges that take place, and the number of inserts the happen.

You might never experience this with `vmod_xkey`. But even if you haven't yet, it could just be a matter of time.

6.3.2 vmod_ykey

`vmod_ykey` is the *Varnish Enterprise* interpretation of what a *secondary key invalidation VMOD* should look like.

Its approach and core concepts are similar to `vmod_xkey`. But it cannot be considered a *v2* of *Xkey*, because they are completely different modules in the way they address *secondary keys*.

Why Ykey?

The name *Ykey* is intended to reflect the fact that while *Ykey* is distinctly different from *Xkey*, it fulfills a similar use case.

Where `vmod_xkey` was bolted onto the *expiry data structure* of *Varnish*, `vmod_ykey` is a proper implementation that is backed by changes in the *Varnish core*.

It is important to know that `vmod_ykey` is only available in *Varnish Enterprise* and that the core changes aren't reflected in the source code of *Varnish Cache*.

`vmod_xkey` is a very *square* module with tons of *sharp edges*. It has very specific rules and not a lot of flexibility. Over time, we received lots of requests to make *Xkey* more flexible, but due to its architecture that was not possible.

The *API* that *Ykey* delivers to interface with the *VMOD* inside *VCL* is also not backwards compatible. As a matter of fact: `vmod_xkey` operates outside of the scope of *VCL*.

The lack of a viable upgrade path for `vmod_xkey` led to the development of `vmod_ykey`.

vmod_ykey performance improvements

Ykey is integrated into the core of *Varnish*, and we specifically made sure it works well with *MSE*.

More specifically, with *persisted MSE caches*.

As you remember from *Xkey*, after every restart, all the persisted objects will be reindexed separately. That would result in tons of *disk I/O*.

`vmod_ykey` is designed to persist the *secondary key index*, not in the *MSE stores*, but in the *MSE books*.

More detail about the *Massive Storage Engine* and its architecture will be presented in the next chapter. But until then, just remember the following two concepts:

The *store* contains the headers and payload of an object. It is stored in a big pre-allocated file on disk.

The *book* is a *metadatabase*, implemented using *LMDB*. It's an embedded database based on *memory-mapped files*.

The fact that *indexed keys* are persisted in a fast but reliable mechanism doesn't just speed up invalidation, it also makes indexing a *one-time cost*.

Indexing doesn't happen automatically when `import ykey;` takes place. The *VCL API* allows for various rules to be defined, which impacts how *secondary key indexing* is done. By default nothing is done until you instruct *Ykey* to do so.

Because `vmod_ykey` doesn't piggyback on the *expiry data structure*, and has its own data structures in the core of *Varnish Enterprise*, the *expiry mechanism* doesn't block all the time due to locking. This results in a smoother flow that doesn't jeopardize regular operations.

Registering keys

As mentioned, `vmod_ykey` behaves in an entirely different way from `vmod_xkey`, especially in terms of indexing. The *API* reflects this.

An interesting concept is that not all *keys* should be registered via *HTTP response headers*:

- `ykey.add_key()` registers an individual key to an object.
- `ykey.add_keys()` registers multiple keys to an object, based on a separator.
- `ykey.add_hashed_keys()` registers multiple keys to an object, based on a separator, with the assumption that they are already hashed.
- `ykey.add_blob()` also registers an individual key to an object, but instead of a *string* value, a *BLOB* value is used to create the hash of the key.

Headers are also supported, just like in `vmod_xkey`, but the *VCL API* allows for a lot more flexibility:

`ykey.add_header()`: registers the header that should be inspected. Multiple keys coming from that header will be registered as keys, based on a separator.

A combined *VCL example* featuring `ykey.add_key()` and `ykey.add_header()` will show you how to implement this:

```
vcl 4.1;

import ykey;

sub vcl_backend_response {
    ykey.add_header(beresp.http.Ykey, ", ");
    ykey.add_header(beresp.http.Xkey, " ");
    if (beresp.http.Content-Type ~ "^image/") {
        ykey.add_key("IMAGE");
    }
}
```

This example will inspect the `Ykey` header from each *HTTP response* and will extract the keys. A *comma space* separator is used for this.

However, we want to remain compatible with *Xkey*, so we're also looking out for the `Xkey` header where a *space* is used as a separator.

Meanwhile, we also tag images automatically if their `Content-Type` response header starts with `image/`. This doesn't require any response header being set.

Invalidating content

Invalidation of content using *Ykey* is quite similar to *Xkey*. The `ykey.purge()` function's *API* is very similar to `xkey.purge()`.

There is no dedicated *soft purge* method in `vmod_ykey`, but the `ykey.purge()` method takes a second argument, which is a boolean. When set to `true` a *soft purge* is done, which sets the *TTL* to zero, but keeps *grace and keep values* as they are.

By default the *soft purge* argument is false.

A vmod_xkey replica

The following example will use the `ykey.purge()` function, and replicate the behavior of the *Xkey* example:

```
vcl 4.1;

import ykey;
import std;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-ykey-purge) {
            return(synth(400,"x-ykey-purge header missing"));
        }

        set req.http.x-purges = ykey.purge(req.http.x-ykey-purge);

        if (std.integer(req.http.x-purges,0) != 0) {
             return(synth(200, req.http.x-purges + " objects
purged"));
        } else {
            return(synth(404, "Key not found"));
        }
    }
}

sub vcl_backend_response {
    ykey.add_header(beresp.http.Ykey, ", ");
}
```

The limitation of this example is that `ykey.purge()` only allows a single key to be invalidated. Luckily `ykey.purge_keys()` can take care of that.

Multiple keys, soft purging

Let's keep the limitations of the previous example in mind, and write an example that can invalidate multiple keys at once. But to switch it up a bit, we'll perform a *soft purge*, which will keep the *grace* and *keeps* settings intact.

This means that the burden of the invalidation is not on the next user. If you paid attention, you'll remember that *grace* will allow users to receive a *stale version* of the object, while *Varnish* asynchronously fetches the new version.

Here's the code to achieve this:

```
vcl 4.1;

import ykey;
import std;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-ykey-purge) {
            return(synth(400,"x-ykey-purge header missing"));
        }

        set req.http.x-purges = ykey.purge_keys(req.http.x-ykey-
purge, ", ", true);

        if (std.integer(req.http.x-purges,0) != 0) {
             return(synth(200, req.http.x-purges + " objects
purged"));
        } else {
            return(synth(404, "Key not found"));
        }
    }
}

sub vcl_backend_response {
    ykey.add_header(beresp.http.Ykey, ", ");
}
```

A quick heads-up here: this example will use a *comma space* separator.

Native support for headers

The previous example was entirely built on the concepts of `vmod_xkey`. Some of the checks aren't required, as `vmod_ykey` has native support for headers through the `ykey.purge_header()` function.

The difference is subtle and can only be felt if you use multiple `x-ykey-purge` headers in a single request.

This is the code:

```
vcl 4.1;

import ykey;
import std;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if(!req.http.x-ykey-purge) {
            return(synth(400,"x-ykey-purge header missing"));
        }

        set req.http.x-purges = ykey.purge_header(req.http.x-ykey-
purge, ", ", true);

        if (std.integer(req.http.x-purges,0) != 0) {
             return(synth(200, req.http.x-purges + " objects
purged"));
        } else {
            return(synth(404, "Key not found"));
        }
    }
}

sub vcl_backend_response {
    ykey.add_header(beresp.http.Ykey, ", ");
}
```

This is an example where the X-Ykey-Purge header has multiple occurrences:

```
PURGE / HTTP/1.1
Host: example.com
X-Ykey-Purge: category_sports
X-Ykey-Purge: category_breaking_news
```

The ykey.purge_header() will loop through all occurrences because the req.http.x-ykey-purge argument is treated as a header type, and is not converted into a string type.

The previous example where we used `ykey.purge_keys()` wouldn't support this because the `req.http.x-ykey-purge` argument is treated as a `string` type, and would only return the first occurrence of the `X-Ykey-Purge` header, which would be `category_sports`.

Namespacing

Another advantage of `vmod_ykey` is *namespace support*. It allows secondary keys to be stored in a namespace to avoid *key collisions* in a *multi-tenant* setup.

Without *namespacing*, multiple independent clients or backends that use the same *Varnish* could risk using the same keys.

The `ykey.namespace()` function allows *key indexing* at the backend level, and *purging* at the client level, to happen in a separate namespace.

The following example injects `ykey.namespace()` calls into `vcl_recv` for the *client-side context*, into `vcl_backend_response` for the *backend-side context*, and resets it when not in the same *namespace*:

```
vcl 4.1;

import ykey;
import std;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if (!req.http.x-ykey-purge) {
            return(synth(400,"x-ykey-purge header missing"));
        }
        if (req.http.host ~ "tenant1") {
            ykey.namespace(req.http.host);
        } else {
            ykey.namespace_reset();
        }
        set req.http.x-purges = ykey.purge_header(req.http.x-ykey-
purge, ", ", true);

        if (std.integer(req.http.x-purges,0) != 0) {
             return(synth(200, req.http.x-purges + " objects
purged"));
        } else {
            return(synth(404, "Key not found"));
        }
    }
}

sub vcl_backend_response {
    if (bereq.http.host ~ "tenant1") {
        ykey.namespace(bereq.http.host);
    } else {
        ykey.namespace_reset();
    }
    ykey.add_header(beresp.http.Ykey, ", ");
}
```

6.4 Forcing a miss

All of the previous *cache invalidation* examples resulted in objects being removed from cache without fetching the updated version of the object.

The burden is always on the next visitor to trigger the revalidation process. The blow is softened when *soft purges* are used because *grace mode* might still trigger a *background fetch* while stale content is temporarily served.

The downside of *soft purges* is that the outdated content is still around for the duration of the revalidation.

All these mechanisms are designed to *delete content* rather than *refresh content*.

However, there is a *VCL variable* available from the client subroutines called `req.hash_always_miss`. By default its value is `false`. But when you set it to `true`, a *cache hit* will be treated as a *cache miss*, and the content will get refreshed. The old version of the object will remain in cache until it expires or is evicted by other invalidation strategies.

We could trigger such a refresh using a custom `REFRESH` *HTTP method*. We could also borrow the *ACL check* from the *purge* example.

We could end up with the following code:

```
vcl 4.1;

backend default {
    .host = "localhost";
    .port = "8080";
}

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "REFRESH") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        set req.hash_always_miss = true;
        set req.method = "GET";
    }
}
```

This *refresh mechanism* will be triggered using the following *HTTP request:*

```
REFRESH / HTTP/1.1
Host: example.com
```

The output would not be a *synthetic message*, but the actual content of the new object.

In terms of integration, you could combine the *hit always miss* logic, with *bans*, or *purges*.

6.5 Distributed invalidation with Varnish Broadcaster

A *ban*, a *purge*, a *refresh*, even *secondary* keys, can easily be triggered via simple *HTTP requests*.

We've discussed the *invalidation mechanisms* that offer the most flexibility when it comes to invalidating multiple objects. But there's a different kind of flexibility that we're still lacking that we haven't talked about.

For really basic setups, *Varnish* can be hosted on the same machine as the origin. But for mission-critical setups, you want some level of *high availability*.

This means that in most setups you'll use more than one *Varnish* server. Although we'll discuss *high availability* in the next chapter, there is one aspect that we need to cover here: *invalidating content on multiple Varnish servers*.

In situations where multiple *Varnish* servers are in play, the client is responsible for sending an invalidation call to each client. But as your *Varnish inventory* increases, keeping track of every server can become challenging, and sending out all those *purge* calls can become equally challenging.

And that's where you need *Varnish Broadcaster*.

6.5.1 Varnish Broadcaster

The *Varnish Broadcaster* comes in the form of the `broadcaster` program and is a utility that is shipped with *Varnish Enterprise*.

As the name indicates, it *broadcasts* messages to a pre-defined inventory of *Varnish* servers. This tool was specifically developed to perform *purges* and *bans* on multiple *Varnish servers* through a single point of entry.

In this section we will treat the *Varnish Broadcaster* as a utility to invalidate the cache across multiple servers, but we will not focus on `broadcaster` itself, and how it is configured. In the next chapter, we will talk more about certain operational elements of *Varnish*, and more in-depth information about `broadcaster` will be covered there.

6.5.2 Varnish inventory

The *broadcaster* cannot figure out on its own where the *Varnish* servers are located. For node discovery, it depends on a `nodes.conf` file where the inventory is specified.

Multiple *Varnish endpoints* can be described in this file, and nodes can be grouped as well.

Imagine the following setup in `nodes.conf`:

```
[eu]
eu-varnish1 = http://varnish1.eu.example.com
eu-varnish2 = http://varnish2.eu.example.com
eu-varnish3 = http://varnish3.eu.example.com

[us]
us-varnish1 = http://varnish1.us.example.com
us-varnish2 = http://varnish2.us.example.com
us-varnish3 = http://varnish3.us.example.com
```

The setup described in the example above consists of two geographic zones:

- An *eu* zone with three *Varnish* servers
- A *us* zone with three *Varnish* servers

By performing a purge call through the *Varnish broadcaster*, the purges will be broadcast to the following *Varnish* servers:

- `http://varnish1.eu.example.com`
- `http://varnish2.eu.example.com`
- `http://varnish3.eu.example.com`
- `http://varnish1.us.example.com`
- `http://varnish2.us.example.com`
- `http://varnish3.us.example.com`

If we just want the *eu* zone to be invalidated, a specific *header* can be sent to the *broadcaster* service. This will limit the scope of the broadcasting.

6.5.3 Issuing a purge

If we want to perform a *purge* on our full inventory, we could send the following request to the *broadcaster*:

```
PURGE / HTTP/1.1
Host: example.com
```

The call itself is identical, but the endpoint we connect to is different:

```
curl -X PURGE example.com:8088/
```

As you can see the *broadcaster endpoint* is hosted on a different port than *Varnish*. Here's the output you get:

```
{
    "method": "PURGE",
    "uri": "/",
    "ts": 1603633688,
    "nodes": {
        "eu-varnish1": 200,
        "eu-varnish2": 200,
        "eu-varnish3": 200,
        "us-varnish1": 200,
        "us-varnish2": 200,
        "us-varnish3": 200
    },
    "rate": 100,
    "done": true
}
```

What you're seeing is *JSON* output with metadata of your request, but also the nodes that were called. All six nodes were purged, and each node returned an *HTTP 200* status.

6.5.4 Bans and secondary keys

Let's add a level of complexity and evict objects from the cache based on a *regular expression pattern*. Under the hood, we use *bans* to achieve this.

The *HTTP request* is as follows:

```
BAN / HTTP/1.1
Host: example.com
X-Ban-Pattern: ^/products/
```

Here's the `curl` implementation:

```
curl -X BAN -H "X-Ban-Pattern: ^/products/" example.com:8088/
```

If we want to invalidate based on *secondary keys*, this will be the request:

```
PURGE / HTTP/1.1
Host: example.com
X-Ykey-Purge: category_sports
```

Here's the `curl` implementation:

```
curl -X PURGE -H "X-Ykey-Purge: category_sports" example.com:8088/
```

And in both cases the output will look the same:

```
{
    "method": "PURGE",
    "uri": "/",
    "ts": 1603652566,
    "nodes": {
        "eu-varnish1": 200,
        "eu-varnish2": 200,
        "eu-varnish3": 200,
        "us-varnish1": 200,
        "us-varnish2": 200,
        "us-varnish3": 200
    },
    "rate": 100,
    "done": true
}
```

6.5.5 Broadcast groups

Because nodes in the `nodes.conf` file can be grouped, it is possible to only broadcast messages to a single group.

Imagine that the content on *Varnish* servers in the *eu* group differs from the *us* group. In this case, it sometimes makes sense to only invalidate a specific group of servers.

Let's throw in an example where we want to invalidate all files in the `/images` folder for the *eu* group:

```
BAN / HTTP/1.1
Host: example.com
X-Ban-Pattern: ^/images/
X-Broadcast-Group: eu
```

This is how you execute this via `curl`:

```
curl -X BAN -H "X-Ban-Pattern: ^/images/" -H "X-Broadcast-Group: eu"
example.com:8088/
```

The output will be slightly different and will only feature responses from *eu* nodes:

```
{
    "method": "PURGE",
    "uri": "/",
    "ts": 1603654040,
    "nodes": {
        "eu-varnish1": 200,
        "eu-varnish2": 200,
        "eu-varnish3": 200
    },
    "rate": 100,
    "done": true
}
```

Other than the *group* definition, there are other *X-Broadcast* headers that can be combined and used to define the broadcasting strategy: * `X-Broadcast-Random`: if the value of this header is set to `*`, the broadcaster will only broadcast to one node in each configured group. The node is selected randomly. * `X-Broadcast-InOrder`: If this header is set to `true`, the broadcaster will handle each node one after the other. This is useful for purging multi-layer setups from upstream to downstream. * `X-Broadcast-Skip`: this header blacklists caches as a whitespace-separated list. They will be skipped when processing a group.

6.6 Summary

Achieving a high hit rate is important, and in the first couple of chapters of the book, we focused on that for obvious reasons.

But as you start aggressively caching content, you'll end up in situations where important content updates need to become visible. Waiting for the *TTL* to expire is not an option.

Remember the saying from the start of this chapter:

> There's only one thing worse than not caching enough, and that is caching for too long.

Thanks to the various *cache invalidation mechanisms* that *Varnish* offers, you can customize the way you want to evict objects from the cache.

Here we mean mechanisms like simple *purges*, regular expression patterns used for *bans*, and even tag-based invalidation using *secondary keys*.

Varnish has got you covered, and *VCL* will allow you to implement invalidation the way you want. But even without *VCL*, you can *ban* content from the cache: `varnishadm` allows direct interaction with the *ban list*.

And for those who don't want to depend on *VCL*, and want to remotely *ban* content, you can still use the *CLI protocol*.

A lot of options, a lot of potential integrations: choose wisely, and integrate *cache invalidation* into your application, so that you never face a situation where *breaking news* doesn't break at all.

Chapter 7: Varnish for operations

Welcome to *chapter 7*, we've come a long way. Up until this point, we've introduced you to the core concepts of *Varnish*, and a lot of functional aspects.

One could say that most of the content catered to the needs of developers so far: getting objects into the cache, deciding when not to, and evicting them when necessary.

But when we talk about running *content delivery services* at scale, getting started with *VCL* is the easy part. Making sure both your *Varnish* servers and the *origin* remain stable, despite millions of requests hitting your platform, that is the real challenge.

In this chapter, we'll switch to a more operational point of view, and focus on other responsibilities. Responsibilities that are more in the wheelhouse of *system administrators* and *IT operations*. Dare I say *DevOps*?

We'll be covering topics like security, high availability, load balancing, monitoring, storage, encryption, deployments, and much more.

Enjoy the ride!

7.1 Install and configure

If you're planning to install *Varnish*, you have quite a number of options as to how you're going to do that.

The first decision you'll have to make is which version of *Varnish*:

- Are you going for a stable release like *Varnish Cache 6.0 LTS*?
- Or do you prefer one of the fresh releases like *Varnish Cache 6.6*?
- Or do you want to use *Varnish Enterprise 6*?

After that, you'll have to select the platform where *Varnish* is going to run:

- Are you planning to install it using *on-premise virtual machines or physical servers*?
- Do you prefer using one of our *official machine images in the cloud* ?
- Or would you rather run the *official Varnish Docker image*?

If you're planning to install *Varnish* on *on-premise infrastructure*, you'll also need to decide what operating system to use.

Let's break this down into individual parts.

7.1.1 Packages

Although you are free to compile *Varnish* from source, it's safe to use the following quote:

> Ain't nobody got time for that.

There are *Varnish* packages available for the following *Linux distributions*:

- Ubuntu
- Debian
- CentOS
- Red Hat
- Fedora

It's a matter of running `apt-get install varnish` or `yum install varnish` depending on the package manager that your distribution supports.

Official packages

The preferred way to go is by installing *Varnish's official packages*, which are maintained by *Varnish Software*.

You can find them at https://packagecloud.io/varnishcache. There are packages for *stable* versions of *Varnish*, and for *fresh* releases of *Varnish* that happen twice per year. There are even packages for *end-of-life* versions of *Varnish*.

My advice is to install the *6.0 LTS* packages because they are the most stable. You can find them at https://packagecloud.io/varnishcache/varnish60lts.

The following Linux distribution versions are supported:

- Debian 9 *(Stretch)*
- Debian 10 *Buster*
- Ubuntu 18.04 LTS *(Bionic)*
- Ubuntu 20.04 LTS *(Focal)*
- Enterprise Linux 7 and 8 *(for CentOS, Red Hat, and Fedora)*

Loading the right repository is quite easy. For *Debian and Ubuntu* systems, you can run the following script:

```
curl -s https://packagecloud.io/install/repositories/varnishcache/
varnish60lts/script.deb.sh | sudo bash
```

For *Red Hat, CentOS, and Fedora* systems, the following script can be used:

```
curl -s https://packagecloud.io/install/repositories/varnishcache/
varnish60lts/script.rpm.sh | sudo bash
```

Both scripts will identify your Linux distribution, register the repository endpoints, verify the *GPG key*, and update the channels.

And in the end, it's a matter of running `apt-get install varnish` or `yum install varnish`.

We would advise you to install the latest version, but if you're interested in which versions are available, you can install a specific version.

On *Debian and Ubuntu*, you can run `apt-cache policy varnish` to see the available versions:

```
$ apt-cache policy varnish
varnish:
  Installed: (none)
  Candidate: 6.0.8-1~bionic
  Version table:
     6.0.8-1~bionic 500
        500 https://packagecloud.io/varnishcache/varnish60lts/ubuntu
bionic/main amd64 Packages
     6.0.7-1~bionic 500
        500 https://packagecloud.io/varnishcache/varnish60lts/ubuntu
bionic/main amd64 Packages
     6.0.6-1~bionic 500
        500 https://packagecloud.io/varnishcache/varnish60lts/ubuntu
bionic/main amd64 Packages
```

On *Red Hat, CentOS, an Fedora*, you can run `yum --showduplicates list varnish` to see the available versions:

```
$ yum --showduplicates list varnish
Available Packages
varnish.x86_64 6.0.6-1.el7 varnishcache_varnish60lts
varnish.x86_64 6.0.7-1.el7 varnishcache_varnish60lts
varnish.x86_64 6.0.8-1.el7 varnishcache_varnish60lts
```

Because reverse compatibility is ensured, it doesn't make much sense to install an older version. However, we've shared this information just so you know that these older versions exist.

Varnish Enterprise packages

Varnish Enterprise packages are installed via https://packagecloud.io/varnishplus, but you need a key to access this repository. These keys come with your *Varnish Enterprise* license.

Once the package channel has been registered on your system, you can run `apt-cache policy varnish-plus` or `yum --showduplicates list varnish-plus` to figure out which versions are available.

Here's the output for *Debian and Ubuntu*:

```
$ apt-cache policy varnish-plus
varnish-plus:
  Installed: (none)
  Candidate: 6.0.8r1-1~bionic
  Version table:
     6.0.8r1-1~bionic 500
        500 https://packagecloud.io/varnishplus/60/ubuntu bionic/main
amd64 Packages
     6.0.7r3-1~bionic 500
        500 https://packagecloud.io/varnishplus/60/ubuntu bionic/main
amd64 Packages
     6.0.7r2-1~bionic 500
        500 https://packagecloud.io/varnishplus/60/ubuntu bionic/main
amd64 Packages
...
```

Here's the output for *CentOS, Red Hat, and Fedora*:

```
$ yum --showduplicates list varnish-plus
...
varnish-plus.x86_64 6.0.7r2-1.el7 varnish-plus-60
varnish-plus.x86_64 6.0.7r3-1.el7 varnish-plus-60
varnish-plus.x86_64 6.0.8r1-1.el7 varnish-plus-60
```

A quick disclaimer about the listing of available packages: we have shortened the list so we don't fill pages with somewhat irrelevant content. Keep in mind that a lot of versions are available.

Then it's just a matter of executing `apt-get install varnish-plus` or `yum install varnish-plus` to install *Varnish Enterprise*.

There are also some other packages you might want to install when running *Varnish Enterprise*:

- `varnish-plus-vmods-extras`: a collection of enterprise *VMODs* that require external dependencies. They are kept separate to keep the initial footprint of *Varnish Enterprise* small.
- `varnish-broadcaster`: the *Varnish Broadcaster* is primarily used to perform distributed cache invalidations and for *Varnish High Availability*.
- `varnish-plus-discovery`: an *autodiscovery* tool that automatically provisions the broadcaster's `nodes.conf` file

- `varnish-plus-ha`: the *High Availability* component of *Varnish Enterprise*
- `varnish-custom-statistics`: the *Varnish Custom Statistics (VCS)* server that stores custom time-series data that was logged in *VCL*
- `varnish-custom-statistics-agent`: the agent software that sends custom statistics from the *VSL* to the *VCS server*
- `varnish-plus-addon-ssl`: contains *Hitch*, the TLS proxy that is used to support *TLS/SSL* in *Varnish*

Depending on your setup, you're not going to install all these packages on a single server. Especially the *VCS server*, which can be installed on a separate machine.

Distro packages

Various Linux distributions offer packages for *Varnish Cache*. The versions vary a lot, there's no flexibility, and only recent versions of *Debian*, *Ubuntu*, and *CentOS/Red Hat/Fedora* offer packages for *Varnish Cache 6*.

Here's an overview of the distributions and versions that support *Varnish Cache 6* at the time of writing:

- Debian 10 *Buster*: *Varnish Cache 6.1.1*
- Ubuntu 20.04 LTS *(Focal)*: *Varnish Cache 6.2.1*
- Enterprise Linux 8 *(for CentOS, Red Hat, and Fedora)*: *Varnish Cache 6.0.2* via `epel-release`

If you're planning to install *Varnish* on *CentOS*, *Red Hat*, or *Fedora*, you must install `epel-release`, which is done via `yum install -y epel-release`.

For the operating systems mentioned above, it's just a matter of running `apt-get install varnish` or `yum install varnish` to install *Varnish*.

We discourage you from installing these packages, as they aren't updated. You will probably get some security fixes when there's a vulnerability, but regular bugfixes and new features are only available if you use our official packages.

7.1.2 Cloud images

Deploying *Varnish* in the cloud is an easy way to get a *full-blown Varnish setup* without having to install and configure the software yourself.

Most public cloud vendors have a marketplace where *machine images* are advertised. *Varnish Software* is also on these marketplaces and offers official machine images.

These images are available on the following cloud platforms:

- Amazon Web Services
- Microsoft Azure
- Google Cloud Platform
- Oracle Cloud Infrastructure
- DigitalOcean

Amazon Web Services

Information on the various images on *AWS* is available at https://aws.amazon.com/marketplace/seller-profile?id=263c0208-6a3a-435d-8728-fa2514202fd0.

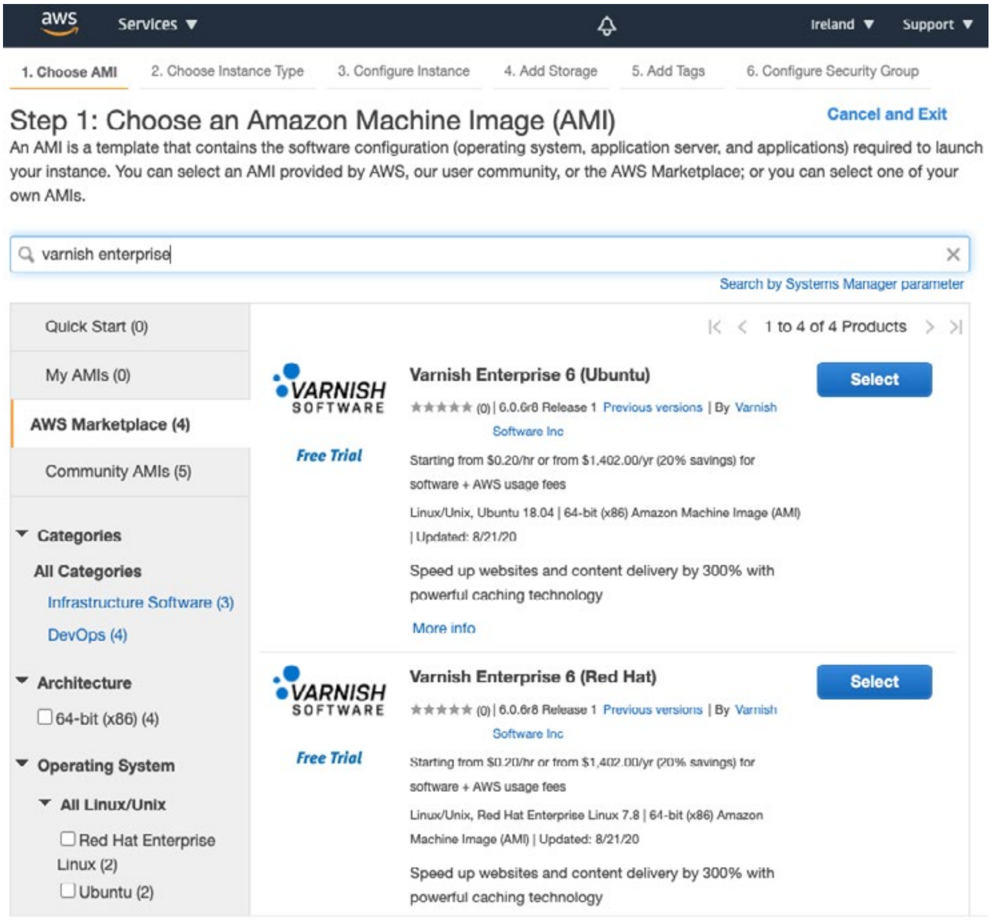
Images that are worth mentioning:

- Varnish Enterprise 6 for Ubuntu
- Varnish Enterprise 6 for Red Hat

When spinning up *virtual machines* that use this image, you'll end up with a fully functional *Varnish* server that was automatically set up and configured and only needs your *VCL*.

A small management fee is charged by the hour, and is separately billed by *AWS* on top of the infrastructure cost. This fee also includes the license of *Varnish Enterprise*. This is great because you can actually start using *Varnish Enterprise* without an upfront license cost. However, you don't get a *Service Level Agreement* and the same level of support.

The image below shows how you can select our images from the *AWS Marketplace*:



Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

varnish enterprise

Search by Systems Manager parameter

Quick Start (0)
My AMIs (0)
AWS Marketplace (4)
Community AMIs (5)

Categories
All Categories
Infrastructure Software (3)
DevOps (4)

Architecture
64-bit (x86) (4)

Operating System
All Linux/Unix
Red Hat Enterprise Linux (2)
Ubuntu (2)

1 to 4 of 4 Products

Varnish Enterprise 6 (Ubuntu)
★★★★★ (0) | 6.0.6r8 Release 1 Previous versions | By Varnish Software Inc
Starting from $0.20/hr or from $1,402.00/yr (20% savings) for software + AWS usage fees
Linux/Unix, Ubuntu 18.04 | 64-bit (x86) Amazon Machine Image (AMI) | Updated: 8/21/20
Speed up websites and content delivery by 300% with powerful caching technology
More info

Varnish Enterprise 6 (Red Hat)
★★★★★ (0) | 6.0.6r8 Release 1 Previous versions | By Varnish Software Inc
Starting from $0.20/hr or from $1,402.00/yr (20% savings) for software + AWS usage fees
Linux/Unix, Red Hat Enterprise Linux 7.8 | 64-bit (x86) Amazon Machine Image (AMI) | Updated: 8/21/20
Speed up websites and content delivery by 300% with powerful caching technology

AWS marketplace

When you spin up a new *EC2 instance*, the *Quick Start* option is selected in *step 1*. If you click *AWS Marketplace* and search for *Varnish Enterprise*, you'll find some related images that you can spin up.

When the server is *up-and-running*, and you access it via *HTTP*, you'll get a welcome page that directs you to https://info.varnish-software.com/cloud/new-instance for further instructions.

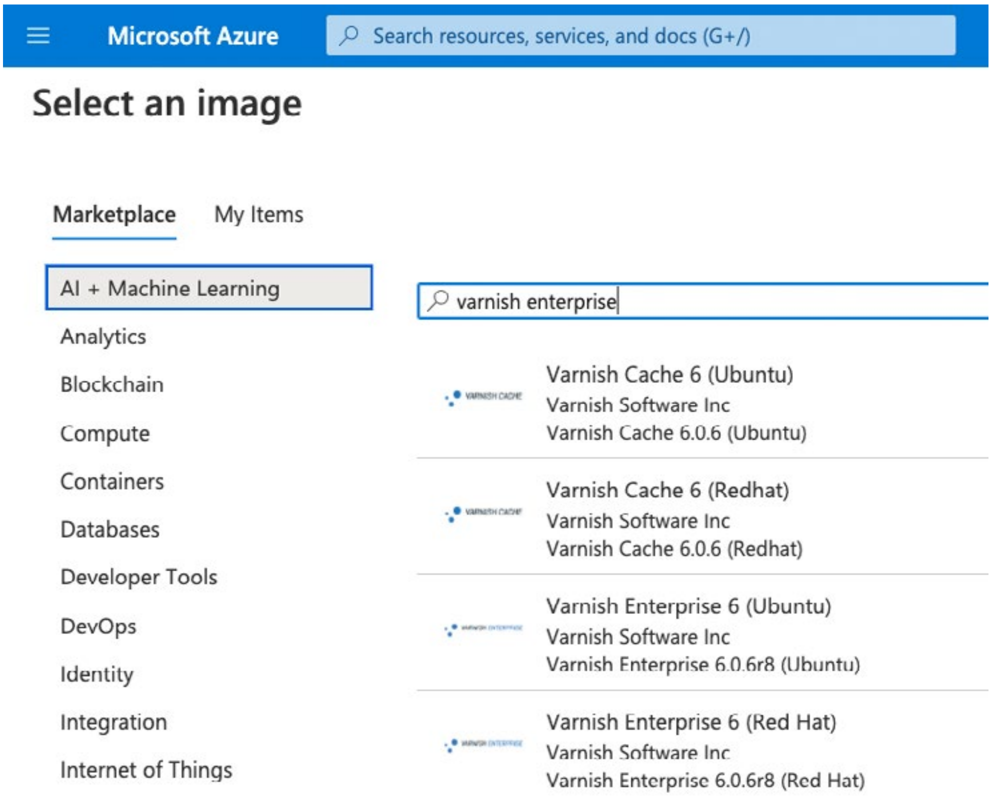
Microsoft Azure

Varnish Enterprise images are also available on the *Microsoft Azure Marketplace* at https://azuremarketplace.microsoft.com/en-us/marketplace/apps/varnish.varnish-cache_.

The same licensing deal applies on *Azure*: *Microsoft* will send you a bill for the infrastructure cost and will bill you separately for the licensing. This will also allow you to use *Varnish Enterprise* without an upfront license cost.

In a similar fashion, when spinning up a virtual machine on *Azure* and selecting the *machine image*, you click *Browse all public and private images*.

The image below shows what this image browser looks like. When you search for *Varnish Enterprise*, this is the result you get:



Varnish Cache 6 (Ubuntu)
Varnish Software Inc
Varnish Cache 6.0.6 (Ubuntu)

Varnish Cache 6 (Redhat)
Varnish Software Inc
Varnish Cache 6.0.6 (Redhat)

Varnish Enterprise 6 (Ubuntu)
Varnish Software Inc
Varnish Enterprise 6.0.6r8 (Ubuntu)

Varnish Enterprise 6 (Red Hat)
Varnish Software Inc
Varnish Enterprise 6.0.6r8 (Red Hat)

Azure marketplace

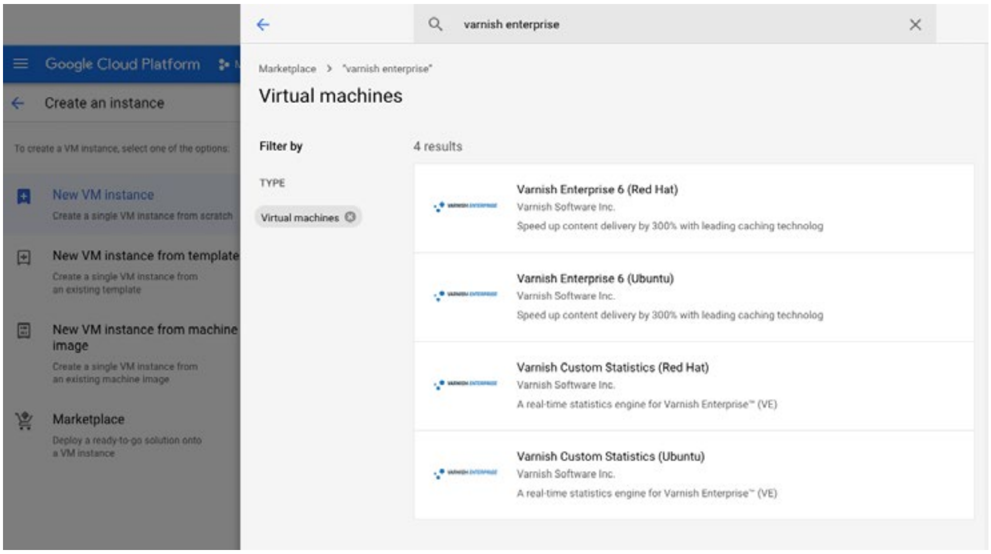
Once the machine is *up-and-running*, and you access it via *HTTP*, you'll also be directed to https://info.varnish-software.com/cloud/new-instance for further instructions.

Google Cloud Platform

Google Cloud Platform (GCP) also has a marketplace and also features our official machine images. You'll find them at https://console.cloud.google.com/marketplace/browse?q=varnish%20software%20inc..

In terms of cost and payment, it’s the same deal as for *AWS* and *Azure*: *Google* charges you for the infrastructure and bills you separately for the licensing.

When you’re launching a new *virtual machine*, and you want to use one of our images on *GCP*, you’ll be given the option to create a *New VM instance*. Please click *Marketplace instead*. When you search for *Varnish Enterprise*, this is what you get:


Google Cloud Platform

Create an instance

To create a VM instance, select one of the options:

New VM instance
Create a single VM instance from scratch

New VM instance from template
Create a single VM instance from an existing template

New VM instance from machine image
Create a single VM instance from an existing machine image

Marketplace
Deploy a ready-to-go solution onto a VM instance

varnish enterprise

Marketplace > "varnish enterprise"

Virtual machines

Filter by

4 results

TYPE

Virtual machines

Varnish Enterprise 6 (Red Hat)
Varnish Software Inc.
Speed up content delivery by 300% with leading caching technolog

Varnish Enterprise 6 (Ubuntu)
Varnish Software Inc.
Speed up content delivery by 300% with leading caching technolog

Varnish Custom Statistics (Red Hat)
Varnish Software Inc.
A real-time statistics engine for Varnish Enterprise™ (VE)

Varnish Custom Statistics (Ubuntu)
Varnish Software Inc.
A real-time statistics engine for Varnish Enterprise™ (VE)

GCP marketplace

And again, once the machine is *up-and-running*, and you access it via *HTTP*, you’ll also be directed to https://info.varnish-software.com/cloud/new-instance for further instructions.

Oracle Cloud Infrastructure

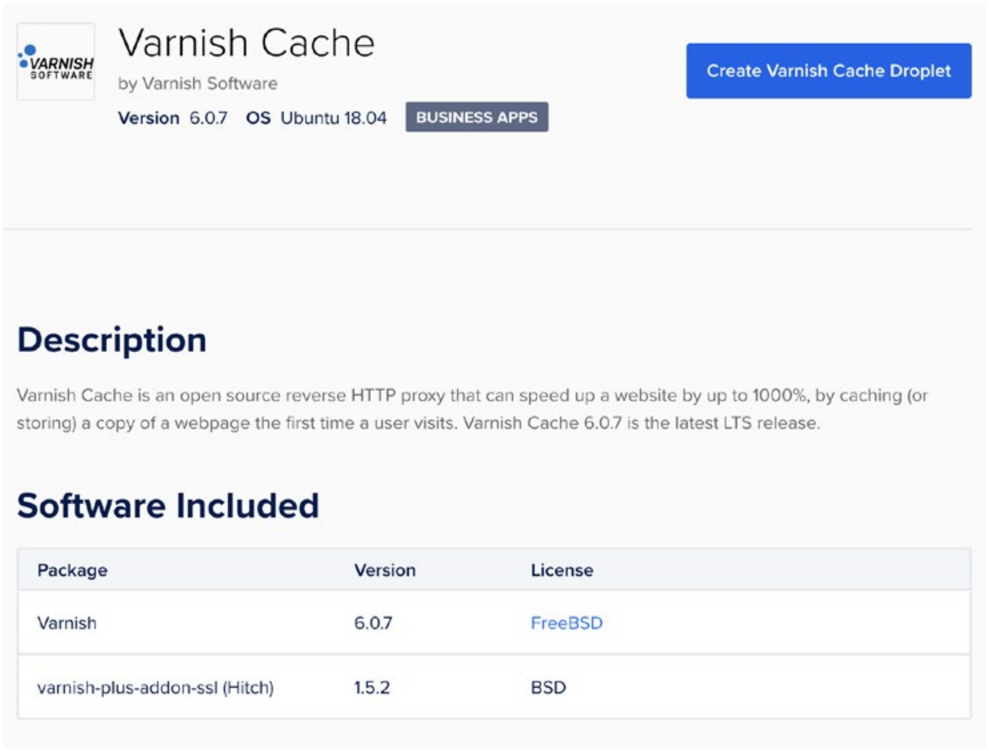
Oracle Cloud Infrastructure is a different animal. In terms of licensing they apply a *Bring Your Own License (BYOL)* policy. And on *OCI* we only support *Varnish Enterprise* on a custom *Oracle Linux distribution*.

More information is available at https://cloudmarketplace.oracle.com/marketplace/en_US/listing/73388855.

DigitalOcean

We offer an official *DigitalOcean* droplet for *Varnish Cache* on their marketplace. See https://marketplace.digitalocean.com/apps/varnish-cache for more information.

Because this is an open source image, no license fees are charged. The benefit of using the droplet, compared to installing *Varnish Cache* yourself, is that the droplet features *Varnish Cache 6.0 LTS*. This is the *long-term support* version that is maintained by *Varnish Software*. At the time of writing, this is version `6.0.8`.



Varnish Cache

by Varnish Software

Version 6.0.7 **OS** Ubuntu 18.04

Description

Varnish Cache is an open source reverse HTTP proxy that can speed up a website by up to 1000%, by caching (or storing) a copy of a webpage the first time a user visits. Varnish Cache 6.0.7 is the latest LTS release.

Software Included

Package	Version	License
Varnish	6.0.7	FreeBSD
varnish-plus-addon-ssl (Hitch)	1.5.2	BSD

DigitalOcean marketplace

7.1.3 Official Docker container

Varnish is also available for Docker, and there is an official *Varnish Cache Docker image*, which is available on the *Docker Hub* at https://hub.docker.com/_/varnish.

We support two kinds of images for *Varnish Cache*:

- *Stable* releases that refer to our *6.0 LTS* release
- *Fresh* releases that refer to the latest release, which is currently version *6.6*

Unless you need *fresh features*, I'd advise you to use *stable* releases. Here's how you would run the official image using the `docker run` command:

```
docker run --name varnish -v /path/to/varnish/default.vcl:/etc/var-
nish/default.vcl:ro --tmpfs /var/lib/varnish:exec -p 80:8080 -d var-
nish:stable
```

This command does the following:

- Launch a container named `varnish`
- Mount the file `/path/to/varnish/default.vcl` on your host system onto `/etc/varnish/default.vcl` in the container as a *read-only file*
- Mount the `/var/lib/varnish` directory of the container in `tmpfs`, which means the contents will be stored in *memory*, and only *exec* calls are allowed
- Forward the exposed port `80` on the container to port `8080` on the host system
- Daemonize the command using the `-d` parameter
- Load the `varnish:stable` image, which is the `stable` tag that currently refers to *Varnish Cache 6.0.6 LTS*

And in the end you can access *Varnish* by calling `http://localhost:8080` in your browser.

There is also an *official Docker image for Varnish Enterprise*, but this image is not public, and only accessible to users who have a *Varnish Enterprise* license key.

These Docker containers can be run *standalone*, but they can also be orchestrated with `docker-compose`, *Kubernetes*, or other *cloud-native* orchestration software.

7.1.4 Kubernetes

If you want to run *Varnish* in *Kubernetes*, you can use our official *Docker*. Unfortunately, at the time of writing there is no official *Varnish Helm chart*.

However, for your convenience, we have created a *Kubernetes* configuration.

Config map definition

The first part of the *Kubernetes* configuration we'll create is the config map. We use it to store our *VCL* configuration.

Here's the `ConfigMap` definition:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: varnish
  labels:
    name: varnish
data:
  default.vcl: |+
    vcl 4.1;
    backend default none;

    sub vcl_recv {
        if (req.url == "/varnish-ping") {
            return(synth(200));
        }

        if (req.url == "/varnish-ready") {
            return(synth(200));
        }

        return(synth(200,"Welcome"));
    }
```

The name of this `ConfigMap` is `varnish` and the `name` label is also `varnish`. Inside the config map we store a key named `default.vcl` which contains the *VCL file*. You'll notice we used an oversimplified *VCL config* to limit the size of the configuration.

Service definition

The next thing we need is a *service definition*. It allows the *Kubernetes pods* to be exposed to the outside world.

```
apiVersion: v1
kind: Service
metadata:
  name: varnish
  labels:
    name: varnish
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: varnish-http
  selector:
    name: varnish
```

In terms of naming and labeling, we stuck with `varnish`. This service will bind itself to port `80` on the IP address of the *Kubernetes cluster*.

Deployment definition

The deployment is the part of the configuration that refers to the containers. In *Kubernetes* containers are run in *pods*. Containers inside these pods share the network.

Here's our simplified deployment configuration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: varnish
  labels:
    name: varnish
spec:
  replicas: 1
  selector:
    matchLabels:
      name: varnish
  template:
    metadata:
      labels:
        name: varnish
    spec:
      containers:
        - name: varnish
          image: "varnish:stable"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /varnish-ping
              port: 80
            initialDelaySeconds: 30
            periodSeconds: 5
          readinessProbe:
            httpGet:
              path: /varnish-ready
              port: 80
            initialDelaySeconds: 30
            periodSeconds: 5
          volumeMounts:
          - name: varnish
```

```
            mountPath: /etc/varnish
      volumes:
      - name: varnish
        configMap:
          name: varnish
```

For naming and labeling we stick with `varnish`. Labels are referred to by selectors in other resource types. In the service definition, the `selector` referred to a *pod* that had the `name: varnish` label.

That way we can link *services* to *pods*. The pod we're creating in our deployment has a *Varnish* container named `varnish` that uses the `varnish:stable` *Docker* image. This is our official image.

Port `80` is exposed and used by the service through the service's `targetPort:80` configuration.

The deployment also refers to a *liveness probe* and a *readiness probe*. These URL endpoints are monitored by *Kubernetes*.

The `livenessProbe` configuration is used to check whether the container is still alive. If the endpoint doesn't respond, the container is restarted. The `readinessProbe` configuration is used to decide whether or not the container can accept traffic.

You may have noticed that these probing endpoints were defined in the *VCL file*. As long as they return the synthetic *HTTP 200* response, all is good.

There's also a volume definition. It creates a volume named `varnish` that refers to the *config map* we created. This way, we can mount the *config map* as a disk inside the container. Via the `volumeMounts` configuration in the container definition, we can mount the config map to the `/etc/varnish` folder. This results in `/etc/varnish/default.vcl` being available.

Deploying Varnish to Kubernetes

Assuming that the `ConfigMap` definition, the `Service` definition, and the `Deployment` definition are all stored in separate *YAML* files in the same folder, the following command can be used to deploy them to *Kubernetes*:

```
$ kubectl apply -f .
configmap/varnish created
deployment.apps/varnish created
service/varnish created
```

The `kubectl get all` is one of those commands that can be run to figure out the status of the resources we just created:

```
$kubectl get all
NAME                          READY   STATUS    RESTARTS   AGE
pod/varnish-dbc8dbc9c-zfc8t   1/1     Running   0          84s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
service/varnish      ClusterIP   10.96.39.230   <none>        80/TCP
84s

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/varnish   1/1     1            1           84s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/varnish-dbc8dbc9c   1         1         1       84s
```

The service is bound to port `80` on IP address `10.96.39.230` of your cluster.

If you're running *Kubernetes* on your local machine, you can use `kubectl port-forward` to forward the port of a service to a local port on your computer:

```
$ kubectl port-forward service/varnish 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

And in the end, we can access our service via `http://localhost:8080` and receive our synthetic output:

```
$ curl http://localhost:8080
<!DOCTYPE html>
<html>
  <head>
    <title>200 Welcome</title>
  </head>
  <body>
    <h1>Error 200 Welcome</h1>
    <p>Welcome</p>
    <h3>Guru Meditation:</h3>
    <p>XID: 2</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
```

If you want to clean up after yourself, just run `kubectl delete -f .` to delete the created resources from your *Kubernetes* server.

An important disclaimer is that this example is not necessarily an accurate representation of what is required to run *Varnish* inside *Kubernetes* in a production environment. When it comes to backend selection, logging, persistent storage, high availability and autoscaling, there is a lot more to be done.

But that, ladies and gentlemen, is beyond the scope of this book.

7.2 Configuring Varnish

Once *Varnish* is installed, you will need to configure the `varnishd` program using a set of options and *runtime parameters*. When *Varnish* is installed via packages, in the cloud, or on Docker, conservative default values are set for you.

Chances are that these defaults aren't to your liking and will need to be tweaked. An overview of all options and parameters can be found at http://varnish-cache.org/docs/6.0/reference/varnishd.html.

In this subsection, we'll talk about common parameters and how you can modify their values.

7.2.1 Systemd

When *Varnish* is installed via packages, or in the *cloud*, the *systemd* service manager will be used to run `varnishd` and to provide configuration options.

The configuration for the *Varnish service* can be found in `/lib/systemd/system/varnish.service`:

```
[Unit]
Description=Varnish Cache, a high-performance HTTP accelerator
After=network-online.target

[Service]
Type=forking
KillMode=process

# Maximum number of open files (for ulimit -n)
LimitNOFILE=131072

# Locked shared memory - should suffice to lock the shared memory log
# (varnishd -l argument)
# Default log size is 80MB vsl + 1M vsm + header -> 82MB
# unit is bytes
LimitMEMLOCK=85983232

# Enable this to avoid "fork failed" on reload.
TasksMax=infinity

# Maximum size of the corefile.
LimitCORE=infinity

ExecStart=/usr/sbin/varnishd -a :6081 -f /etc/varnish/default.vcl -s
```

```
malloc,256m
ExecReload=/usr/sbin/varnishreload

[Install]
WantedBy=multi-user.target
```

What you can see is that *Varnish* is listening on port `6081` for incoming connections, that the `/etc/varnish/default.vcl` *VCL file* is loaded, and that *256 MB* of memory is allocated for object storage.

The unit files in `/lib/systemd/system/`are not to be edited. Instead, *systemd* allows you to override these files by creating appropriate files in `/etc/systemd/system/`.

Editing via systemctl edit

If you want to modify some of these settings, you can run the following commands:

```
sudo systemctl edit varnish
```

Here's an example where we set the object allocation to *512 MB* instead of the standard *256 MB* value.

```
[Service]
ExecStart=
ExecStart=/usr/sbin/varnishd -a :6081 -f /etc/varnish/default.vcl -s
malloc,512m
```

Please note that you need to explicitly clear `ExecStart` before setting it again, as it is an additive setting.

This will create `/etc/systemd/system/varnish.service.d/override.conf`, which will not interfere with package upgrades.

To view the *unit* file including the override:

```
$ sudo systemctl cat varnish

# /etc/systemd/system/varnish.service
[Unit]
Description=Varnish Cache, a high-performance HTTP accelerator
After=network-online.target

[Service]
Type=forking
KillMode=process

# Maximum number of open files (for ulimit -n)
LimitNOFILE=131072

# Locked shared memory - should suffice to lock the shared memory log
# (varnishd -l argument)
# Default log size is 80MB vsl + 1M vsm + header -> 82MB
# unit is bytes
LimitMEMLOCK=85983232

# Enable this to avoid "fork failed" on reload.
TasksMax=infinity

# Maximum size of the corefile.
LimitCORE=infinity

ExecStart=/usr/sbin/varnishd -a :6081 -f /etc/varnish/default.vcl -s
malloc,256m
ExecReload=/usr/sbin/varnishreload

[Install]
WantedBy=multi-user.target

# /etc/systemd/system/varnish.service.d/override.conf
[Service]
ExecStart=
ExecStart=/usr/sbin/varnishd -a :6081 -f /etc/varnish/default.vcl -s
malloc,512m
```

Restart *Varnish* to make changes take effect: `sudo systemctl restart varnish`.

7.2.2 Docker

The `varnishd` configuration for our official *Docker* container doesn't use *systemd*. It is *Docker* that runs the `varnishd` process in the foreground of the container.

The *Dockerfile* uses an *entrypoint file* to define how *Varnish* should run. This is what it looks like:

```
varnishd \
        -F \
        -f /etc/varnish/default.vcl \
        -a http=:80 \
        -a proxy=:8443,PROXY \
        -p feature=+http2 \
        -s malloc,$VARNISH_SIZE \
        "$@"
```

- The `varnishd` program is started in the foreground thanks to the `-F` option.
- `varnishd` will listen for incoming connections on port `80` for regular *HTTP*, and this listening port is named `http`.
- `varnishd` will listen for incoming connections on port `8443` for *HTTP* using the *PROXY protocol*, and this listening port is named `proxy`.
- `HTTP/2` is supported thanks to the `-p feature=+http2` parameter.
- `varnishd` will allocate a fixed amount of memory to object storage. The size is defined by the `$VARNISH_SIZE` environment variable, which defaults to `100M`.
- Any additional runtime parameter that is added in the `docker run` command will be attached to `varnishd`, thanks to `"$@"`.

The minimal configuration required to run this *Docker container* is done using the following command:

```
docker run -p 80:80 varnish
```

Let's say you want to override the `default.vcl` file, name the container `varnish`, set the cache size to `1G`, make the `default_ttl` an hour, and reduce the `ban_lurker_age` to *ten seconds*. This is the command you'll use for that:

```
docker run --name varnish -p 80:80 \
        -v /path/to/default.vcl:/etc/varnish/default.vcl:ro \
        -e VARNISH_SIZE=1G\
        varnish \
        -p default_ttl=3600 \
        -p ban_lurker_age=10
```

Certain aspects of the configuration are handled by *Docker*:

- The `-p` parameter allows you to forward the exposed ports of the container to ports on your host system.
- The `-v` parameter allows you to perform a *bind mount* and make a local *VCL file* available in the container.
- The `-e` parameter allows you to set an *environment variable*. In this case the `VARNISH_SIZE` variable is set to `1G`.

Thanks to `"$@"` in the *entrypoint file*, all additional positional arguments will be attached to the `varnishd` process. This means that you can just add any supported *Varnish runtime parameter* to `docker run`, which will be added to `varnishd`.

In this case we added `-p default_ttl=3600` and `-p ban_lurker_age=10`, which will translate into `varnishd` runtime parameters. This provides enormous flexibility and doesn't require the creation of custom images.

7.2.3 Port configuration

We've featured the `-a` option a number of times, but there is still a lot to be said about the *listening address* option in *Varnish*. If `-a` is omitted, `varnishd` will listen on port `80` on all interfaces.

Here's the syntax for `-a`:

```
-a <[name=][address][:port][,PROTO][,user=<user>][,group=<group>]
[,mode=<mode>]>
```

- The `name=` field allows you to name your *listening addresses*.
- The `address` part allows you to define an *IPv4* address, an *IPv6* address, or the path to a *Unix domain socket (UDS)*.
- The `:port` part allows you to set the port on which this address is supposed to listen.
- The `PROTO` field defines the protocol that is used; by default this is `HTTP`, but it can also be set to `PROXY` to support the *PROXY protocol*.
- When a *UDS* is used, fields like `user`, `group`, and `mode` are used to control and define permissions on the socket file.
- Multiple `-a` *listening addresses* can be used.
- Unnamed *listening addresses* will be automatically named `a0`, `a1`, `a2`, etc.

Let's throw in an example configuration that uses *(nearly)* all of the syntax:

```
varnishd -a uds=/var/run/varnish.sock,PROXY,user=varnish,group=var-
nish,mode=660 \
         -a http=:80 \
         -a proxy=localhost:8443,PROXY
```

Let's break this one down:

There is a listening address named `uds` that listens for incoming requests over a *Unix domain socket*. The socket file is `/var/run/varnish.sock` and is accessible to the `varnish` user and `varnish` group. Because of `mode=660`, the `varnish` user has *read and write access*, as do all users in the `varnish` group. All other users have no access to the socket file. The protocol that is used for communication over this *UDS* is the *PROXY protocol*.

There is also a listening address named `http`, which accepts regular *HTTP* connections for all interfaces on port `80`.

And finally, there's a listening address named `proxy` that only accepts connections on the `localhost` loopback interface over port `8443`. And again, the *PROXY protocol* is used.

This setup is often used when *Hitch* is installed on the server to terminate *TLS*. Regular *HTTP* connections directly arrive on *Varnish*. But *Hitch* takes care of *HTTPS* requests and forwards the decrypted data to *Varnish* using *HTTP* over the *PROXY protocol*.

Hitch can either choose to connect to *Varnish* using a *Unix domain socket (UDS)*, or via `localhost` over *TCP/IP* on port `8443`.

7.2.4 Object storage

The `-s` option defines how `varnishd` will store its objects. If the option is omitted, the `malloc` storage backend will be used, which stores objects in memory. The default storage size is *100M*.

A pretty straightforward example is one where we assign *1G* of memory to *Varnish* for object storage:

```
varnishd -s malloc,1G
```

Naming storage backends

You can also name your storage backends, which makes it easier to identify them in *VCL* or in `varnishstat`. Here's how you do that:

```
varnishd -s memory=malloc,1G
```

If you don't name your storage backends, *Varnish* will use names like `s0`, `s1`, `s2`, etc.

If an object is larger than the *memory size*, you'll see the following errors appear in the `varnishlog` output:

```
ExpKill         LRU_Fail
FetchError      Could not get storage
```

Varnish notices that there is not enough space available to store the object, so it starts to remove the *least recently used (LRU)* objects. This action fails because there is not enough space in the cache to free up.

Additionally, the full content cannot be fetched from the backend and stored in cache, hence the `Could not get storage` error.

In the end, the object will only be partially served.

Transient storage

It might sound surprising, but there's also a secondary storage backend in use. It's called *transient storage* and holds *short-lived objects*.

Varnish considers an object *short-lived* when its `TTL + grace + keep` is less than the `shortlived` runtime parameter. By default this is *ten seconds*.

Transient storage is also used for temporary objects. An example is uncacheable content that is held there until it is consumed by the client. This is to avoid letting a slow client occupy a backend for too long.

By default, *transient storage* uses an *unlimited* `malloc` backend. This is something that should be kept in mind when sizing your *Varnish* server.

However, *transient storage* can be limited by adding a *storage backend* that is named `Transient`.

Here's an example:

```
varnishd -s Transient=malloc,500M
```

In this example, we're limiting *transient storage* to *500M*.

Limiting the *transient storage* can negatively affect short-lived objects. If whatever is delivered is bigger than the *transient storage* size, objects will only be partially delivered, as they don't fully fit when streaming is enabled. When streaming is disabled, it will lead to an *HTTP 503* error.

File storage

There is also *file storage* available in *Varnish*. This type of object storage will store objects in memory backed by a file.

This is initiated as specified below:

```
varnishd -s file,/path/to/storage,100G
```

In this case, objects are stored in `/path/to/storage`. This file is `100G` in size.

Although disk storage is used for this kind of *object storage*, the `file` *stevedore* is not persistent. A restart will empty the entire cache.

The performance of this *stevedore* also varies quite a lot, as you depend on the write speed of your disk. As your `varnishd` process runs, you will incur an increasing amount of fragmentation on disk, which will further reduce the performance of the cache.

Our advice is to not use the `file` *stevedore* at a large scale, and use the *MSE stevedore* instead.

MSE

The *Massive Storage Engine (MSE)* is a *Varnish Enterprise stevedore* that combines memory and disk storage to offer fast and persistent storage.

We will talk about *MSE* in detail in one of the next sections. Let's limit this discussion to configuration.

Here's how you set up *MSE*:

```
varnishd -s mse,/var/lib/mse/mse.conf
```

As you can see the `mse` stevedore can refer to a configuration file that holds more details about the *MSE* configuration.

Here's what such a configuration file can look like:

```
env: {
    id = "mse";
    memcache_size = "5G";

    books = ( {
        id = "book";
        directory = "/var/lib/mse/book";
        database_size = "2G";

        stores = ( {
            id = "store";
            filename = "/var/lib/mse/store.dat";
            size = "100G";
        } );
    } );
};
```

This configuration will allocate `5G` of memory for object storage. There is also `100G` of persistent storage available, which is located in `/var/lib/mse/store.dat`.

All metadata for persisted objects is stored in `/var/lib/mse/book`, which is `2G` in size.

Using `vmod_mse`, you can let *VCL* decide where objects should be persisted.

MSE is highly optimized and doesn't suffer from the same delays and fragmentation as the `file` *stevedore*.

If you set `memcachesize = "auto"` in your *MSE* configuration, the *memory governor* will be activated. This will dynamically size your cache, based on the memory it needs for other parts of *Varnish*.

The *memory governor* will also be activated when you haven't specified a configuration file for *MSE*:

```
varnishd -s mse
```

The *memory governor* will not limit the size of the cache, but the total size of the `varnishd` process. The total size is determined by the `memory_target` runtime parameter, which is set to `80%` by default. The `memory_target` can also be set to an absolute value. This is very convenient as it allows you to bound the memory of *Varnish* as a whole and not worry about unexpected overhead.

MSE is one of the most powerful features of *Varnish Enterprise*, but as mentioned, we'll do an *MSE* deep-dive later in this chapter.

7.2.5 Not using a VCL file

Although *VCL* is the most powerful feature of *Varnish*, you can still decide to stick with the *built-in VCL*.

In that case, *Varnish* doesn't know what the *backend host and port* are.

The `-b` option allows you to set this, but it is mutually exclusive with the `-f` option that is used to set the location of the *VCL file*.

Here's how you use `-b`:

```
varnishd -b localhost:8080
```

This example lets *Varnish* connect to a backend that is hosted on the same machine, but on port `8080`.

You can also use a *UDS* to connect, as illustrated below:

```
varnishd -b /path/to/backend.sock
```

7.2.6 Varnish CLI configuration

The *Varnish CLI*, which is accessible via `varnishadm`, or via a socket connection in your application code, has a set of parameters that can be configured.

The `-T` option is the primary `varnishd` option to open up access to the *CLI*. This option defines the *listening address* for *CLI* requests.

Authentication to the `varnishd` *CLI* port is protected with a challenge and a secret key. The location of the secret key is defined by the `-S` parameter.

Here's an all-in-one example containing both options:

```
varnishd -T :6082 -S /etc/varnish/secret
```

Anyone can create a *socket connection* to the *Varnish* server on port `6082`. The secret that is in `/etc/varnish/secret` will be required to satisfy the challenge that the `varnishd` *CLI* imposes.

However, in most cases, you don't actually need to specify `-S` or `-T` because if they are not given, `varnishd` will just generate them randomly. But, in that case, how can `varnishadm` know about these parameters? Very simply, if `varnishadm` isn't given a `-S`/`-T` combination, it'll look at the `varnishd` workdir to figure those values out.

The workdir is the value of the `-n` parameter, which defaults to `/var/lib/varnish/$HOSTNAME`. This is why, in the default case, `varnishadm` needs no extra parameters to access the *Varnish CLI*. The `-S` and `-T` parameters are mainly there to configure external access.

7.2.7 Runtime parameters

Besides the basic `varnishd` startup options, there is a large collection of *runtime parameters* that can be extended.

The full list can be found at http://varnish-cache.org/docs/6.0/reference/varnishd.html#list-of-parameters.

It is impossible to list them all, but a very common example is enabling *HTTP/2*. Here's how you do that:

```
varnishd -p feature=+http2
```

These are feature flags, but we can also assign values. For example, we can redefine what are considered *short-lived objects*, by setting the `shortlived` runtime parameter:

```
varnishd -p shortlived=15
```

This means that objects with a *TTL* less than *15 seconds* are considered *short-lived* and will end up in *transient storage*.

By adding runtime parameters to the *systemd* `override.conf` file for *Varnish*, you can persist the new value of these parameters. It is also possible to set them via `varnishadm param.set` at runtime, but these aren't persisted, and will be lost upon the next restart.

7.3 TLS

7.3.1 Historically

Historically, *Varnish* didn't support *TLS/SSL*.

The primary focus has always been *performance*, and back when *Varnish* was coming up, the use of *SSL* and *TLS* wasn't as prevalent as it is now.

With performance in mind, there was always a fear that the implementation of a *crypto* layer would jeopardize performance.

There was also the *UNIX philosophy* lingering in the back of the minds of *Varnish's creators*:

> Do one thing well.

The one thing *Varnish* does well is caching. And if *TLS/SSL* is so important, should it be part of *Varnish*, or should it be implemented in another layer of your *web platform*?

At some point, *Varnish* went for the latter, but decided to facilitate *TLS termination* by supporting the *PROXY protocol*.

This way *Varnish* could still focus on its core duty, which is caching, but the threshold for *TLS termination* could be lowered thanks to the *PROXY protocol*.

At this point, the *PROXY protocol* should sound familiar. We've covered it a number of times throughout the book. You should also know that `varnishd` can listen for incoming connections over `PROXY`. And thanks to `vmod_proxy`, you can extract *proxy-protocol-v2 TLV attributes* from the *TCP connection*.

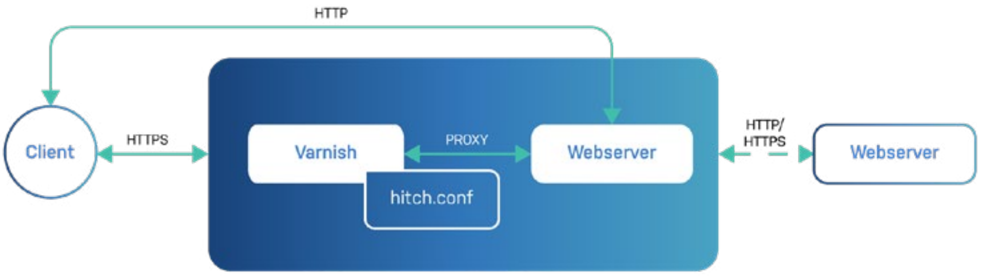
7.3.2 Hitch

Lowering the barrier for *TLS termination* was the ambition at first, but in 2015 *Varnish Software* decided to contribute to this by developing an open source *TLS proxy*.

In the vein of the *UNIX philosophy*, the goal was to create a lightweight *TLS proxy* that was built for the job, but that would also play nice with *Varnish*.

The project was named *Hitch*, and unsurprisingly the one thing it does well is TLS.

Hitch is a separate program that you run in front of your *Varnish* server, as illustrated in the diagram below:



Hitch

What you also see is that *Hitch and Varnish* are often installed on the same server. They communicate with each other over the *PROXY protocol*. This can be done over regular *TCP/IP*, but a *UNIX domain socket (UDS)* connection can also be made to further reduce latency.

Although *Varnish* did lower the barrier for *TLS termination*, there was always a risk that adding an extra hop in the form of a *third-party TLS proxy* could have an impact on performance and latency.

By introducing *Hitch* as a component in the reference architecture of *Varnish*, we could meet guarantees of performance, scalability, and low latency. The throughput rate of *100 Gbps* on a single server is proof of that.

Hitch was specifically designed to terminate *TLS* connections for *Varnish*, but it does not exclusively work with *Varnish*. It is not even restricted to *HTTP traffic*.

Any service that communicates over *TCP/IP* and that requires *TLS* can be terminated using *Hitch*.

Installing Hitch

The project's website can be found at https://hitch-tls.org/, but the code itself is hosted on https://github.com/varnish/hitch.

You are free to compile *Hitch* from source; there's no denying that packages are a lot more convenient. Up until recently, you had to rely on *distro packages*, which are often outdated.

More recently, we decided to provide official packages for *Hitch* at https://packagecloud.io/varnishcache/hitch. This happens to be the same place where official *Varnish Cache* packages can be found. This makes installing these packages look familiar.

For *Debian and Ubuntu* systems, you can run the following script:

```
curl -s https://packagecloud.io/install/repositories/varnishcache/
hitch/script.deb.sh | sudo bash
```

For *Red Hat, CentOS, and Fedora* systems, the following script can be used:

```
curl -s https://packagecloud.io/install/repositories/varnishcache/
hitch/script.rpm.sh | sudo bash
```

And in the end you either run `apt-get install hitch`, or `yum install hitch`, depending on your Linux distribution.

Configuring Hitch

After installing *Hitch*, you can customize its behavior by modifying `/etc/hitch/hitch.conf`. In order to activate these changes, you have to reload the `hitch` process via *systemd*:

```
sudo edit /etc/hitch/hitch.conf
sudo systemctl reload hitch
```

You're not solely reliant on `/etc/hitch/hitch.conf`. The `hitch` program also has a number of command line parameters that can be used to extend the default behavior or to override settings that were defined in `hitch.conf`.

But even when using a configuration file, `hitch` will need to know where to find it, so you'll use the `--config` parameter to indicate that:

```
hitch --config=FILE
```

But let's talk about *Hitch* configurations. We've categorized some interesting ones into five groups, which represent the five next subsections.

Networking settings

Hitch is a proxy server, just like *Varnish*. This means we need to configure how it accepts connections, and how it proxies them through to *Varnish*.

The *listening address* is configured using a `frontend` block. As you can see in the example below, accepting connections on all interfaces on port `443` is the most common use case:

```
frontend = {
    host = "*"
    port = "443"
}
```

You can have multiple frontend blocks in a single configuration file, and these frontend blocks can hold additional settings.

Here's an example where we have two frontends, each with their own certificates:

```
frontend = {
    host = "1.2.3.4"
    port = "443"
    pem-file = "/etc/hitch/example.com.pem"
}

frontend = {
    host = "5.6.7.8"
    port = "443"
    pem-file = "/etc/hitch/foo.com.pem"
}
```

Just so you know, this is just a hypothetical example. You don't really need to define multiple frontends to host multiple certificates. *SNI* will make sure you can serve multiple certificates on the same endpoint by inspecting the *Subject Alternative Name* of the certificate.

The `frontend` can also be defined as an one-liner, as illustrated below:

```
frontend = "[*]:444"
```

And it is also possible to attach a certificate to the `frontend`:

```
frontend = "[*]:444+/etc/hitch/cert.pem"
```

We still need to talk about *backend connections*. The `backend` configuration can be used to define where *Hitch* is going to proxy its traffic to.

Here's an example where we connect to *Varnish* over *TCP/IP*:

```
backend = "[127.0.0.1]:8443"
```

But we can also connect using a *UDS*:

```
backend = "/var/run/varnish.sock"
```

And before we talk about the next subject, I'd like to show you an example where `frontend` and `backend` configurations are done via *command line arguments* and not via the configuration file:

```
sudo hitch -u hitch -f "[*]:443+/etc/hitch/cert.pem" -b
"[127.0.0.1]:80"
```

The `-u` parameter defines which user should be used to run `hitch`. The `root` user is not allowed.

When the `backend` directive or command line argument refers to a hostname, this hostname is resolved to the corresponding IP address upon startup. By default this happens only once. When the hostname is changed, and resolves to another IP address, *Hitch* does not notice this, and keeps on sending data to the IP address it resolved upon startup.

Hitch has a `backend-refresh` setting that allows you to define the frequency of the *backend DNS resolution*. The default value is zero, meaning no backend refresh takes place.

The following example will allow backend DNS refreshes to happen every 30 seconds:

```
backend-refresh = 30
```

And this is the command line equivalent:

```
sudo hitch -u hitch -f "[*]:443+/etc/hitch/cert.pem" -b "[backend.
example.com]:80" -R 30
```

The `-R` option is the shorthand for `--backend-refresh`, which is also supported.

Certificate settings

We just showed you that you can bind certificates to *frontend listening addresses*. But in most cases, it makes more sense to define the *certificate location* with a global scope.

The `pem-file` directive can be added to your Hitch configuration file outside of the `frontend` data structure:

```
pem-file = "/etc/hitch/example.com.pem"
```

The *PEM file* refers to a *x509 certificate file*. It is a concatenation of the *private key*, the main *certificate*, and the corresponding *certificate chain*.

The `pem-file` directive can be used multiple times to load multiple certificates. *Server Name Indication (SNI)* will make sure the right certificate is loaded based on *Subject Alternative Name* of the certificate.

However `pem-file` is more than a *one-liner*, it is a data structure of its own. It allows you to split up the certificate from the private key.

Here's an example of a `pem-file` definition with a separate private key file:

```
pem-file = {
    cert = "/etc/hitch/example.com.pem"
    private-key = "/etc/hitch/private.key"
}
```

This data structure can also be defined multiple times. *SNI* will again make sure the right certificate is matched.

If you have a bunch of certificate files that happen to change occasionally, there is a more flexible way to define them, which is by using the `pem-dir` directive.

Its value is a directory where certificates are stored. Here's an example:

```
pem-dir = "/etc/hitch/cert.d"
```

Hitch will iterate through that directory and load the certificate information from every file.

It does make sense to define a fallback `pem-file` directive in your configuration, in case you are dealing with clients that do not support *SNI*. If you do not define a fallback, the first match of the iteration of the `pem-dir` will be used. If you set `sni-nomatch-abort = off`, the connection will abort if *SNI* didn't find a matching certificate.

There is also a `pem-dir-glob` directive that allows you to be more selective when using `pem-dir` to load certificates from a directory. A *glob* pattern can be applied while iterating through the directory.

When you combine them, you end up with the following configuration:

```
pem-dir = "/etc/hitch/cert.d"
pem-dir-glob = "*.pem"
```

This example will load all certificates from the `/etc/hitch/cert.d` folder that match the `*.pem` pattern. A certificate like `/etc/hitch/cert.d/example.com.pem` would be matched, whereas `/etc/hitch/cert.d/example.com.cert` wouldn't.

Loading certificates is also possible through *command line arguments*. We've already seen an example where a certificate was bound to a frontend. Let's look at an example where a certificate is defined at the global scope:

```
sudo hitch -u hitch -f "[*]:443" -b "[127.0.0.1]:80" /etc/hitch/cert.
pem
```

Protocol settings

When it comes to protocols, there are three questions to be asked:

- Which *TLS/SSL* protocols are we using to accept connections?
- What protocol are we using to connect to the backend?
- Which *application-layer protocols* are we going to announce to the clients?

TLS protocols

Let's start off by saying that *SSL is dead*. For the sake of clarity, we talk about *TLS/SSL*, but in reality we're no longer using the *SSL protocol*. It's all *TLS*.

The `tls-protos` directive allows us to set the *TLS protocols* that *Hitch* is willing to support. And although it is technically possible to mention `SSLv3` as a potential protocol, the best way to configure this directive nowadays is as follows:

```
tls-protos = TLSv1.2 TLSv1.3
```

> The support protocol versions also depend on what the *OpenSSL* version on your server supports. For `TLSv1.3`, *OpenSSL version 1.1.1* is the minimum requirement.

The `--tls-protos` command line option can be used to override whatever is stored in your configuration file. Here's an example of how to define the *TLS protocols* via the command line:

```
sudo hitch -u hitch \
           -f "[*]:443" -b "[127.0.0.1]:80" \
           --tls-protos="TLSv1.3" /etc/hitch/cert.pem
```

PROXY protocol

When we talk about proxying requests to the backend, we've been consistently talking about *HTTP*. That is not the case in *Hitch* because it has no awareness of *HTTP*.

What we can do is enable the *PROXY protocol*, and share information about the original client connection. This information is sent as a *transport-layer header* as soon as the connection is established, and does not depend on any *layer 7 protocol*.

Enabling the *PROXY protocol* can be done via the `write-proxy-v2` directive, as illustrated below:

```
write-proxy-v2 = on
```

Of course, if your backend only supports *version 1* of the protocol, you can set it as follows:

```
write-proxy-v1 = on
```

There is also a `proxy-proxy` setting that is used when incoming connections to *Hitch* are also made using the *PROXY protocol*. In that case the incoming *PROXY* information is proxied to the backend.

Here's an example of how you enable this:

```
proxy-proxy = on
```

The `write-proxy-v1`, `write-proxy-v2`, and `proxy-proxy` directives are all mutually exclusive with one another.

And of course, these settings can also be configured via the command line. Here's the example that proves it:

```
sudo hitch -u hitch -f "[*]:443" -b "[127.0.0.1]:80" \
    --write-proxy-v2="on" /etc/hitch/cert.pem
```

ALPN protocols

Hitch is not aware of any *layer 7* protocol, but *TLS* offers an extension to negotiate which *application-layer* protocol is about to be used.

The extension is called *ALPN*, which is short for *Application-Layer Protocol Negotiation*. The client announces the protocols it supports, and *Hitch* can then choose which *application-layer protocol* it supports.

ALPN is commonly used to check whether or not the server supports *HTTP/2*.

It seems a bit contradictory that a service like *Hitch*, which has no notion of *layer 7 protocols*, is actually getting involved in the *layer 7 protocol negotiation*.

But it's all being done in the name of efficiency: because the negotiation is part of the *TLS protocol*, no extra roundtrips are required. And after all, apart from announcing *application-layer protocols* it supports, it does nothing with it afterwards.

Although *Hitch* is not strictly tied into *HTTP*, the primary use case of *Hitch* is terminating *TLS* for *HTTPS* connections.

Here's an example where we configure *ALPN protocols* via the `alpn-protos` directive:

```
alpn-protos = "h2, http/1.1"
```

When the client supports `HTTP/2`, this setting will result in an `HTTP/2` connection being established. *Hitch* will terminate the *TLS* connection, and shuffle the rest of the bytes to *Varnish*. Because `h2` was part of the *ALPN list*, the client will assume that *Varnish* supports `HTTP/2`.

Therefore it is important to attach the `-p feature=+http2` runtime parameter to `varnishd`, otherwise the client will attempt to send `HTTP/2` traffic to *Varnish* even though it doesn't support it.

Cipher settings

If `TLSv1.3` were used as a TLS protocol, the `ciphersuites` directive would be used to determine which cryptographic algorithms are used.

Here's the default value:

```
ciphersuites = "TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_
CHACHA20_POLY1305_SHA256"
```

Although cryptography itself is very much outside of the scope of this book, it's worth explaining what some of these values mean. Take for example `TLS_AES_128_GCM_SHA256`.

This uses an *Advanced Encryption Standard with 128bit key in Galois/Counter mode* as the encryption algorithm. The hash that is used to ensure the authenticity of the encryption is a *SHA256* hash.

These algorithms are exclusive to `TLSv1.3` and have no overlap with other TLS versions. If you're on `TLSv1.2` or older, you can use the `ciphers` directive to describe the accepted cryptographic algorithms:

```
ciphers = "ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:-
ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECD-
SA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-
SHA256:DHE-RSA-AES256-GCM-SHA384"
```

By specifying both `ciphersuites` and `ciphers`, you remain compatible with both `TLSv1.2` and `TLSv1.3`. The reason for these two settings being separate is that `TLSv1.3` brought with it a completely new set of cipher suites, none of which are compatible with older versions of *TLS*.

We advise sticking with the default values, and when in doubt, have a look at https://wiki.mozilla.org/Security/Server_Side_TLS.

You can also choose who is in charge of selecting the used ciphers. The `prefer-server-ciphers` directive is responsible for that.

This is the default value:

```
prefer-server-ciphers = off
```

This means the client chooses which cipher is used. If you set it to `on`, *Hitch* will choose. In that case the order of the specified ciphers is significant: a cipher specified early will take precedence over the ones specified later.

OCSP stapling

OCSP is short for *Online Certificate Status Protocol* and is a protocol that checks the revocation status of TLS certificates. *OCSP* will check the status of the certificate by performing an HTTP call to the *Certificate Authority's OCSP server*. The corresponding URL is stored inside the certificate. Although *OCSP* is a lot more efficient than its predecessor, the *Certificate Revocation List (CRL)* protocol, which downloaded a list of revoked certificates, it has potential performance implications.

Not only is there added latency for the end-user because of the *OCSP* call to the *CA* for every TLS connection, it also puts a lot of stress on the *OCSP* servers. These servers could end up serving millions of requests at the same time and might crumble under the heavy load. Unverified *OCSP* calls result in errors being displayed in the browser.

What is OCSP stapling?

OCSP stapling is an alternative mechanism that shifts the responsibility for the *OCSP call* from the client to the server. This means that the server will perform occasional *OCSP* calls for the certificates it manages on behalf of the client. The resulting status is *stapled* onto the *TLS handshake*, and removes the need for a client to perform an *OCSP* call.

This means fewer roundtrips, less latency for the client, and less stress on the *OCSP servers*. When a client doesn't receive a stapled response, it may perform the *OCSP* call itself.

Although it may appear that *OCSP stapling* can allow servers to falsely verify fraudulent certificates, there are security mechanisms in place to prevent this. The *OCSP* response is securely signed by the *CA* and cannot be tampered with by the server. If the signature verification doesn't match, the client will abort the connection.

OCSP support in Hitch

Hitch supports *OCSP stapling* and has some configuration directives to control certain aspects of this mechanism.

The `ocsp-dir` directive is the directory in which *OCSP* responses are cached. The default directory is `/var/lib/hitch`:

```
ocsp-dir = "/var/lib/hitch"
```

The lifetime of a cached *OCSP* response is determined by whether the refresh information is part of the response. When no such information is provided in the response, *Hitch* will refresh the status of a certificate with a certain frequency. This frequency is determined by the `ocsp-refresh-interval` directive. By default this is *1800 seconds*.

Here's an example of how to lower the value to *500 seconds*:

```
ocsp-refresh-interval = 500
```

When connecting to an *OCSP server*, the `ocsp-connect-tmo` and the `ocsp-resp-tmo` should be respected. These settings represent the *connect timeout and last byte timeout* for these connections. Their respective default values are *4 seconds* and *10 seconds*.

Here's an example in which we add some leniency by increasing the timeouts:

```
ocsp-connect-tmo = 6
ocsp-resp-tmo = 15
```

The stapled *OCSP response* is to be verified by the client, but by enabling `ocsp-verify-staple`, *Hitch* will also verify the response and remove the staple when the verification fails. It is up to the client to perform the *OCSP* check itself.

Mutual TLS

Mutual TLS (mTLS) is a concept where both the server and the client must guarantee their respective identities via *TLS certificates*. For client authentication the same *X.509* standard will be used to issue client certificates.

It is up to the client to present the certificate when connecting to the server using *TLS*. The server can then verify the validity of that certificate, based on a *CA chain* that is stored on the server.

Hitch supports *mTLS* and offers two configuration directives to enable this:

```
client-verify = required
client-verify-ca = "/etc/hitch/certs/client-ca.pem"
```

This example requires the client to authenticate itself using a client certificate. This is done by setting `client-verify = required`. This means if the client doesn't provide a certificate, or the certificate verification fails, the connection will be aborted.

If `client-verify = optional` is set, a client that does not send a client certificate will still be allowed to connect. But if a client sends a certificate that cannot be verified, the connection will be aborted.

The default value is `client-verify = none`, which means no client verification is required.

The `client-verify-ca` parameter refers to a file where the *certificate chain* is stored. The server will use this chain of certificates to verify the incoming client certificate.

Here's an example of how to use client certificates with `curl`:

```
curl --cacert ca.crt \
     --key client.key \
     --cert client.crt \
     https://example.com
```

In this example, `curl` will connect to `https://example.com`, which may require the client to authenticate itself using a certificate.

- The `--cacert` parameter allows the client to send the *certificate chain*. These are trusted certificates that the actual certificate depends on.
- The `--key` parameter contains the location to the *private key*. This key is used to sign the certificate.
- The `--cert` parameter refers to the *actual certificate* that is used for authentication.

Please note that the `--cacert` parameter in `curl` and the `client-verify-ca` configuration directive in `hitch` refer to the same *certificate chain*.

7.3.3 vmod_proxy

When *Hitch* is used to terminate *TLS* connections in front of *Varnish*, it isn't easy for *Varnish* to know whether or not the incoming request was an *HTTPS* request or a regular *HTTP* request.

There are some *VCL* tricks you can use to figure it out.

If you don't use the *PROXY* protocol, you can check the value of the `X-Forwarded-For` header, and if the corresponding IP address matches the one from *Hitch*, you know you're dealing with an *HTTPS* connection. This isn't 100% reliable because the IP address might also be `localhost`, which doesn't tell you a lot.

If you enable the *PROXY* protocol, you can run the following *VCL* code to check the server port:

```
vcl 4.1;

import std;

sub vcl_recv {
    if (std.port(server.ip) == 443) {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

But there's an even better way, which is by leveraging `vmod_proxy`, as we talked about in *chapter 5*:

```
vcl 4.1;

import proxy;

sub vcl_recv {
    if (proxy.is_ssl()) {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

This example uses the `proxy.is_ssl()` to figure out whether or not the connection was encrypted via *TLS*. And there's a lot more information available that can be retrieved from the *PROXY* protocol.

Here's an example where we extract a variety of so-called *TLV attributes* from the *PROXY protocol header*:

```
vcl 4.1;
import proxy;

sub vcl_deliver {
    set resp.http.alpn = proxy.alpn();
    set resp.http.authority = proxy.authority();
    set resp.http.ssl = proxy.is_ssl();
    set resp.http.ssl-version = proxy.ssl_version();
    set resp.http.ssl-cipher = proxy.ssl_cipher();
    set resp.http.client-has-cert-sess = proxy.client_has_cert_
sess();
    set resp.http.client-has-cert-conn = proxy.client_has_cert_
conn();
}
```

Let's talk about the individual functions that were used, and the output they can return:

- `proxy.alpn()` will return the *ALPN token* that was negotiated. This will typically be one of `h2` or `http/1.1`.
- `proxy.is_ssl()` will return a boolean. If the connections were made using *TLS* it will be `true`, and it will be `false` otherwise.
- `proxy.ssl_version()` will return the *TLS/SSL* version that was used during the session. `TLSv1.3` is a common version that can be returned.
- `proxy.ssl_cipher()` will return the *encryption cipher* that was used to set up the encrypted connection. `TLS_AES_256_GCM_SHA384` could be a possible value.
- `proxy.authority()` will return the server name as it was presented by the client during handshake. `example.com` is a possible value.
- `proxy.client_has_cert_sess()` will return a boolean. This function will return `true` if the *client certificate* was provided for the session. It is possible that the session was resumed on a new connection, and that the handshake happened in a previous connection.
- `proxy.client_has_cert_conn()` will also return a boolean. This function will return `true` if the *client certificate* was provided for the connection.

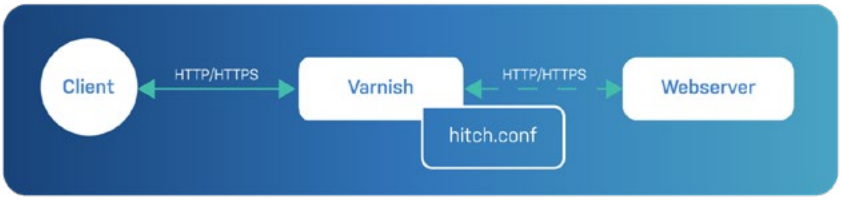
vmod_proxy has some other functions as well, but the *TLV attributes* that are retrieved in these functions are not processed by *Hitch*.

7.3.4 Native TLS in Varnish Enterprise

Although *Hitch* is simple, flexible, stable, secure, and very fast, *Varnish Software* noticed that some of its customers wanted to go beyond the limit of *100 Gbps* per server.

This use case obviously doesn't apply to your average *Varnish* user. But *Varnish Software* decided to find a solution and did so by offering a *native TLS implementation in Varnish Enterprise*.

This *in-core TLS* implementation can easily handle *200 Gbps* on a single server and reuses the configuration syntax of *Hitch*.



Native TLS in Varnish Enterprise

Enabling native TLS

As of *Varnish Enterprise 6.0.6r2* you can use the -A runtime parameter in varnishd, and point it to a *Hitch* config file for native TLS to work.

Here's an example of how to use -A :

```
varnishd -A /etc/varnish/tls.cfg -a :80 -f /etc/varnish/default.vcl
```

Please do not confuse -a with -A:

- -a is used to define *HTTP* and *PROXY* listening addresses.
- -A is used to set up *native TLS*.

Please note that there is no need to define the *TLS port* via -a :443 because this is all done in the *Hitch configuration file*.

Configuring native TLS

As mentioned, *native TLS support in Varnish Enterprise* uses the *Hitch* configuration syntax. It is important to know that it only uses a subset of these configuration directives.

Keep in mind that *native TLS* doesn't need to proxy data through to some backend, so all settings that are related to backend communication are in fact irrelevant. *Varnish* won't complain if these settings are present, but it will ignore them.

Here's a good example of a *TLS configuration file* that can be used in *Varnish*:

```
frontend = {
    host = "*"
    port = "443"
}

pem-file = "/etc/varnish/certs/example.com"

ciphersuites = "TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_
CHACHA20_POLY1305_SHA256"

ciphers = "ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:-
ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECD-
SA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-
SHA256:DHE-RSA-AES256-GCM-SHA384"

tls-protos = TLSv1.2 TLSv1.3

prefer-server-ciphers = false
```

Let's break this configuration down:

- The `frontend` data structure will accept connections on port `443` on all network interfaces.
- The `pem-file` directive points to the certificate that will be used.
- `ciphersuites` is the list of ciphers that will be used for `TLSv1.3` connections.
- `ciphers` is the list of ciphers that will be used for `TLSv1.2` connections.
- `tls-protos` allows `TLSv1.2` and `TLSv1.3` to be used.
- The `prefer-server-ciphers` directive states that the cipher selection can be decided by the client.

When to use native TLS

It seems quite convenient to be able to recycle your *Hitch* configuration file and benefit from *native TLS*.

The *native TLS* feature was exclusively built for performance and to go beyond the *100 Gbps* per server threshold.

Under all other circumstances, we would advise you to continue using *Hitch*. The main reason being security.

Hitch is a separate process that runs under a different user than Varnish. If for some reason there is a *TLS* vulnerability, only the crypto part will be exposed. Your cache, containing potentially sensitive data, will not.

This separation of concerns can also be beneficial from a scalability point of view: you can scale your TLS separately from your caching, and make sure servers are tuned for their respective jobs.

But if you don't care too much about these things, you just might appreciate the fact that *native TLS* removes the need to manage a separate *TLS proxy*. So that's one less service to worry about, and one less thing that can fail on you.

7.3.5 vmod_tls

When using *native TLS in Varnish Enterprise*, the use of `vmod_proxy` becomes irrelevant because there is no proxying happening.

However, there is an alternative: it's called `vmod_tls`, and it has a similar *API*.

Here's the *TLS detection* example from earlier where `proxy` and `ssl` are replaced with `tls`:

```
vcl 4.1;

import tls;

sub vcl_recv {
    if (tls.is_tls()) {
        set req.http.X-Forwarded-Proto = "https";
    } else {
        set req.http.X-Forwarded-Proto = "http";
    }
}
```

This example will set the `X-Forwarded-Proto` request header with the right *scheme* as its value. This allows the application to use the right *scheme* for its URLs, and it can also decide to vary on this header by setting `Vary: X-Forwarded-Proto`.

The other attributes can also be retrieved. Here's an example where we return the various *TLS attributes* via response headers:

```
vcl 4.1;
import tls;

sub vcl_deliver {
    set resp.http.alpn = tls.alpn();
    set resp.http.authority = tls.authority();
    set resp.http.cert-key = tls.cert_key();
    set resp.http.cert-sign = tls.cert_sign();
    set resp.http.tls = tls.is_tls();
    set resp.http.cipher = tls.cipher();
    set resp.http.version = tls.version();
}
```

The output for these headers could be the following:

```
alpn: h2
authority: localhost
cert-key: RSA2048
cert-sign: RSA-SHA256
tls: true
cipher: TLS_AES_256_GCM_SHA384
version: TLSv1.3
```

7.3.6 Backend TLS

We've been focusing so much on *client-side TLS* that we almost forgot to mention that *backend TLS* is also supported. This is a *Varnish Enterprise* feature, by the way.

Configuring backend connections over *TLS* is quite simple, as you can see in the example below:

```
backend default {
    .host = "backend.example.com";
    .port = "443";
    .ssl = 1;
    .ssl_sni = 1;
    .ssl_verify_peer = 1;
    .ssl_verify_host = 1;
}
```

It's just a matter of enabling `.ssl` to turn on *backend TLS*. Keep in mind that you should probably modify the `.port` as well and set it to `443`.

SNI is enabled by default, but the example enables it explicitly.

The `.ssl_verify_peer` option validates the certificate chain of the backend. By disabling this setting, the backend is allowed to use *self-signed certificates*.

`.ssl_verify_host`, which is disabled by default, will check whether or not the `Host` header of the backend request matches the identity of the certificate.

The previously mentioned `vmod_tls` is also of benefit for backend TLS connections. When used from the VCL subroutine `vcl_backend_response`, it will report its values based on the current TLS backend connection, using the exact same interface as described in the previous section.

7.3.7 End-to-end

As we near the end of the *TLS* section of the book, it is clear that *end-to-end* encryption of the connection is possible.

- *Hitch* can be used to terminate *TLS* for *Varnish*.
- *Hitch* also has the ability to enforce *client certificates*.
- *Varnish Enterprise* can provide a *native TLS* implementation.
- *Varnish Enterprise* also has an option to communicate with the backend over *TLS*.

It's safe to say that the crypto part is well-covered.

7.4 Massive Storage Engine

The *Massive Storage Engine* is probably the most significant feature that *Varnish Enterprise* offers. It combines the speed of memory, and the reliability of disk to offer a *stevedore* that can cache *terabytes* of data on a single *Varnish* server.

The cache is persisted and can survive a restart. This means you don't need to rewarm the cache when a *Varnish* server recovers from a restart.

MSE is commonly used to build custom *CDNs* and to accelerate *OTT video platforms*.

But it's not only about caching large volumes of data: *MSE*'s memory implementation represents a clear improvement over *Varnish*'s original `malloc` stevedore.

Before we dive into the details, we need to talk about the history of *MSE*, and why this *stevedore* was developed in the first place.

7.4.1 History

Before *MSE* was a viable *stevedore* for caching large volumes of data, the *Varnish* project had two disk-based *stevedores*:

- `file`
- `persistence`

The file stevedore

The `file` stevedore stores objects on disk, but not in a persistent way: a restart will result in an empty cache.

That is because the `file` stevedore is nothing more than memory storage that is backed by a file: if the size of the cache exceeds the available memory, content will be sent to the disk. And content that is not available in memory will be loaded from the disk.

It's just like the operating system's *swap* mechanism, where the disk is only used to extend the capacity of the memory without offering any persistence guarantees.

The operating system's *page cache* will decide what the *hot* content is that should be buffered in memory. This ensures that not every access to the file system results in *I/O instructions*.

The problem with the `file` stevedore is that the *page cache* isn't really optimized for caches like *Varnish*. The *page cache* will guess what content should be in memory, and if that guess is wrong, file system access is required.

This results in a lot of context switching, which consumes server resources.

The biggest problem, however, is disk fragmentation, which increases over time. Because the `file` stevedore has no mechanism to efficiently allocate objects on disk, objects might be scattered across multiple blocks. This results in a lot more *disk I/O* taking place to assemble the object.

Over time, this becomes a real performance killer. The only solution is to restart *Varnish*, which will allow you to start over again. Unfortunately, this will also empty the cache, which is never good.

The persistence stevedore

As the name indicates, the `persistence` stevedore can persist objects on disk. These objects can survive a server restart. This way your cache remains intact, even after a potential failure.

However, we would not advise you to ever use this *stevedore*. This is a quote from the official *Varnish* documentation for this feature:

> The persistent storage backend has multiple issues with it and will likely be removed from a future version of Varnish.

The main problem is that the `persistence` *stevedore* is designed as a *log-structured file system* that enforces a strict *FIFO* approach.

This means:

- Objects always come in one way.
- Objects always come out the other way.

This means that the first object that is inserted will be the first one to expire. Although this works fine from a *write performance* point of view, it totally sidelines key *Varnish* features, such as *LRU eviction*, banning, and many more.

Since this *stevedore* is basically unusable in most real-life situations, and since the `file` stevedore is not persistent and prone to disk fragmentation, there is a need for a better solution. Enter the *Massive Storage Engine*.

Early versions of MSE

In the very first year of *Varnish Software's* existence, a plan was drawn up to develop a good *file-based stevedore*.

The initial focus was not to develop a *persistent stevedore*, but the critical goal in the beginning was to offer the capability to cache objects that are larger than the available memory.

Even the initial implementation was not that different from the `file` *stevedore*. It was mostly a matter of smoothing the rough edges to create an improved version of the `file` *stevedore*.

Behind the scenes *memory-mapped files* were still used to store the objects. This means that the operating system decides which portions of the file it keeps in the *page cache*. Since the *page cache* corresponds to a part of the system's actual physical memory, the result is that the operating system's *page cache* serves as the memory cache for *Varnish*, storing the *hot objects*, whereas the file system contains all objects.

The critical goal for the *second version of MSE* was to add persistent storage that would survive a server restart. The metadata, residing in memory in *version 1*, also needed to be persisted on disk. *Memory-mapped files* were also used to accomplish this.

A major side effect of *memory-mapped files*, in combination with large volumes of data, is the overhead of the very large memory maps that are created by the kernel: for each page in a memory-mapped file, the kernel will allocate some bytes in its page tables.

If for example the backing files grow to *20 TB* worth of cached objects, an overhead of *90 GB* of memory could occur.

This is memory that is not limited by the storage engine and is expected to be readily available on top of memory that was already allocated to *Varnish* for object storage. Managing these memory maps can become CPU intensive too.

7.4.2 Architecture

With the release of *Varnish Enterprise 6*, a new version of *MSE* was released as well. *MSE 3* was designed with the limitations of prior versions in mind.

Let's look at the improved architecture of *MSE*, and learn why this is such an important feature in *Varnish Enterprise 6*.

Memory vs disk

In previous versions of *MSE* the operating system's *page cache* mechanism was responsible for deciding what content from a *memory-mapped* file would end up in memory. This is the so-called *hot content*.

Not only did this result in large memory maps, which create overhead, it is also tough for the *page cache* to determine which parts from disk should be cached in memory. What is the *hot data*, and how can the *page cache* know without the proper context?

That's the reason why, in *MSE*, we decided to reimplement the *page cache* in *user space*. From an architecture point of view, *MSE version 3* stepped away from *memory-mapped files* for object storage and implemented the logic for loading data from files into memory itself. This also implies that the persistence layer had to be redesigned.

By no longer depending on these *memory-mapped files*, the overhead from keeping track of pages in the kernel has been eliminated.

And because the *page cache* mechanism in *MSE version 3* is a custom implementation, *MSE* has the necessary context and can more accurately determine the *hot content*. The result is higher efficiency and less pressure on the disks, compared to previous versions.

As before, memory only contains a portion of the total cached content, while the persistence layer stores the full cache. However, with the increased control *MSE* has over what goes where, *Varnish* can greatly reduce the number of I/O operations, a limiting factor in many SSD drives. *MSE* can even decide to make an object *memory only* if it realizes it is lagging behind in writing to disk, where previous versions would slow down considerably.

Traditionally, non-volatile storage options have been perceived as slow, but now it is possible to configure a system that can read hundreds of gigabits per second from an array of persistent storage drives. MSE is designed to work well with all kinds of hardware, all the way down to spinning magnetic drives.

As we'll explain next, the disk-based storage layer is comprised of a *metadata database*, which we call a *book*, and the *persistent storage* is something that we call *the store*. You can configure them, use multiple disks to store data on, and have multiple *books*, which can have references to multiple *stores*.

Books

Let's talk about *books* first. As mentioned, these books contain *metadata* for each of the objects in the MSE.

The main questions these books answer are:

- What cached objects does your cache contain at any point in time?
- Where on disk should I look to find these objects?

In terms of *metadata*, the *book* holds the following information:

- Lifetime counters, such as *TTL*, *grace*, and *keep*
- Object hashes
- Information about cache variations
- *Ban* information
- *Ykey* indexes
- Information about free storage
- The location of an object on disk

This information is kept in a *Lightning Memory-Mapped Database (LMDB)*. This is a low-level transactional database that lives inside a *memory-mapped file*. This is a specific case where we still rely on the *operating system's page cache* to manage what lives in memory and what is stored on disk. To keep the database safe to use after a power failure, the database code will *force write* pages to disk.

The size of the *book* is set in the *MSE* configuration, and directly impacts the number of objects you can have in the corresponding *stores*. It does this by imposing a maximum amount of metadata that can reside in the book at any point in time.

If you have few *Ykeys* and your objects are not too big, a good rule of thumb is to have *2 kB book space per object*.

Let's look at a concrete example for this rule of thumb: if you have stores with a total of 10 TB of data, and your average object size is 1 MB, then your maximum number of objects is ten million. With 2 kB per object, the rule indicates that the book should be at least 20 GB.

If it turns out the you have sized the book much bigger than your needs, then the extra space in the book will simply never be used, and its contents will never make it into the *page cache* or consume any memory. If the book becomes too close to full, Varnish will start removing objects for the store, resulting in a potential under-usage of the store. For this reason it is better to err on the safe side when calculating the optimal size for the book unless you have space available to expand the book after the fact.

In most cases, you should account for memory corresponding to the number of bytes *in use* in the book. This will let the kernel keep the book in memory at all times instead of having to *page in* parts of the book that have not been used in a while. The exception is when the book contains a high proportion of objects that are very infrequently accessed, and *paging in* data does not significantly reduce the performance of the system. We will get back to this when discussing the *memory governor* later in this chapter.

It is possible to configure multiple books. This is especially useful when you use multiple disks for cache storage, which improves performance. In case of disk failures, partitioning the cache reduces potential data loss.

The standard location of the books is in `/var/lib/mse`. This is a folder that is allowed by our *SELinux* rules, which is part of our packaging.

What we call the *book* is actually a directory that contains multiple files:

- `MSE.lck` is a lock file that protects the storage from potentially being accessed by multiple *Varnish instances*.
- `data.mdb` is the actual *LMDB* database that contains the metadata.
- `lock.mdb` is the internal lock file from *LMDB*.
- `varnish-mse-banlist-15f19907.dat` is a *per-book ban list*, containing the currently active bans.
- `varnish-mse-journal-75b6069b.dat` is a *per-store* journal that keeps track of incremental changes prior to the final state being stored in `data.mdb`.

Stores

The *stores* are the physical files on disk that contain the cached objects. The *stores* are designed as *pre-allocated large files*. Inside these files, a custom file system is used to organize content.

Stores are associated with a *book*. The *book* holds the location of each object in the *store*. Without the *book*, there is no way to retrieve objects from cache because *MSE* wouldn't know where to look.

> If you lose the *book*, or the *book* gets corrupted, there is no way to regenerate it. That part of your persisted cache is lost. But remember: it's a cache; the data can always be regenerated from the origin.

Every *store* is represented as a single file, and the standard location of these files is `/var/lib/mse`. This is also because our *SELinux* rules allow this folder to be used by `varnishd`.

There are significant performance benefits when using *pre-allocated large files*.

The most obvious one is reducing *I/O overhead* that results from opening files and iterating through directories. Unlike typical storage systems that entirely rely on the file system, *MSE* doesn't create a file per object. Only a single file is opened, and this happens when `varnishd` is started.

Disk fragmentation is also a huge one: by pre-allocating one large file per store, disk fragmentation is heavily reduced. The size of the file and its location on disk are fixed, and all access to persisted objects is done within that file.

MSE also has algorithms to find and create *continuous free space* within that file. This results in fewer *I/O operations* and allows the system to get more out of the disk's bandwidth without being limited by *I/O operations per second (IOPS)*.

Access to *stores* is done using *asynchronous I/O*, which doesn't block the progress of execution while the system is waiting for data from the disks. This also boosts performance quite a bit.

The danger of disk fragmentation

We just talked about the concept of *stores*, and disk fragmentation was frequently mentioned.

A disk is fragmented when single files get spread across multiple regions on the disk. For spinning disks, this would cause a mechanical head to have to seek from one area of the disk to another with a significantly reduced performance as a result. In the era of *SSD*s, disk fragmentation is less of an issue, but it is not free: all drives have a maximum number of I/O operations per second in addition to the maximum bandwidth. When a disk is too fragmented, or objects are very small, this can become a limiting factor. Needless to say, it is important to consider the *I/O operations per second (IOPS)* for the drives when configuring any server, and *NVMe* drives generally perform much better than *SATA SSDs*.

MSE has mechanisms for reducing fragmentation, which work well for most use cases, but huge caches with small objects will still require drives with a high number IOPS.

Selecting a location for the store and book

MSE makes sure that the fragmentation of the store is low, but it cannot control the fragmentation of the store file in the file system. For this reason it is recommended to only put *MSE stores* on volumes dedicated to MSE. The easiest way is to put a single store and its book on each physical drive intended for MSE, and to keep the operating system on a separate drive. When the store file is created, MSE makes sure to pre-allocate all the space for the store file to make sure that the file system cannot introduce more fragmentation after the creation of the store. Some file systems, like `xfs`, do not implement this, so only `ext3` and `ext4` would be candidates. However, we only support `ext4` for the MSE stores.

Currently there is no support for using the block device directly with no file system on top, but it might arrive in the future.

Making sure there is room for more

When an *MSE store or book* starts to get almost full, MSE needs to evict objects for the store or book in question. The eviction process, often called *nuking*, starts when the amount of used space reaches a certain level, explained below. The process tries to delete content that has not been accessed in a while, but the method is slightly different than the *least recently used (LRU)* eviction found in memory-based *stevedores*. The difference is based on MSE's desire to create *continuous free space* to avoid fragmentation.

There are individual *waterlevel* parameters for books and stores, but they both default to *90%*. For the stores, the parameter is called `waterlevel`, while the book's *waterlevel nuking* is controlled by the `database_waterlevel` parameter.

If the parameters are left at the default values, and either the book or the store usage reaches 90%, backend fetches will be paused until the usage goes below 90%. To avoid performance degradation for backend fetches, MSE starts evicting objects before the waterlevel is reached. The runtime parameters `waterlevel_hysterisis` for *stores*, and `database_waterlevel_hysterisis` for *books*, both *defaulting to 5%*, control this behavior. If all the parameters are left at their default values, MSE will start evicting objects when the store or the book are *85% full*, and this is usually sufficient to keep the usage *under 90%*, avoiding stalling fetches as a result.

The goal is to evict neighboring segments within the *store* to create *continuous free space* without removing objects that have been used recently. The thread that is responsible for freeing space by removing objects, scans objects linearly, and tests whether the object is in the third least recently used. If that is the case, it is removed. Once we get below the waterlevel, the eviction mechanism is paused and can resume from that position on the next run.

The description above is actually slightly misleading since we have not yet explained exactly how MSE calculates the amount of *used* and *free* space. Since small free chunks are not usable for placing big objects without creating significant fragmentation, MSE will only consider sufficiently large chunks of unused space when calculating the total number of bytes available for allocation. The parameter `waterlevel_minchunksize` defines what the minimum chunksize is that should be counted, and it defaults to *512 KB*.

In other words, only chunks that are equal or greater than `waterlevel_minchunksize` will be considered when making sure that there is, by default, at least 15% free space in the store. All chunks that are smaller than the *512 KB* default value, remain untouched. However, these smaller chunks of free space are still eligible when MSE needs to insert objects that are smaller than 512 KB, and MSE will even select the smallest one that is big enough for any new allocation.

It might be tempting to reduce the `waterlevel_minchunksize` to a low value, but that will increase fragmentation of bigger objects, as they will often be chopped into pieces equal to the size of `waterlevel_minchunksize`. Such fragmentation actually increases usage in the book, as each chunk will need its own entry in the book.

Basically, `waterlevel_minchunksize` is a tunable fragmentation tolerance level, and finding the right value for you depends on how MSE is used. A high value will minimize fragmentation, which translates into a leaner book and higher performance due to fewer I/O operations, while lower values will fit more small objects into the cache.

Problems with the traditional memory allocator

The `malloc` *stevedore*, used by most *Varnish Cache* servers, needs to be configured to hold a fixed maximum amount of data.

A common rule of thumb is to configure it to be *80%* of the total memory of a server. For a server with *64 GB* of RAM, this translates to a little over *51 GB*. The remaining *20%* is then available for the operating system and for various parts of the `varnishd` process.

Unfortunately, this rule of thumb does not always work well. The optimal value for your server heavily depends on traffic patterns, on *worker thread* memory requirements, on object data structures, and on *transient storage*.

None of these memory needs are accounted for in the `malloc` stevedore, which makes it hard to guess what Varnish's *total memory footprint* will be by just looking at the `malloc` sizing.

The result is that the server will suddenly be out of memory, even when you apply a seemingly conservative size to your `malloc` store. If certain aspects of your *VCL* require a lot of memory to be executed, or if your *transient storage* goes through the roof, you'll be in trouble, and there's no predictable way to counter this. On the other hand, if your server is very simple, stores a few big objects, and does not serve a lot of concurrent users, many gigabytes of memory can sit unused when it would be better to use that memory for caching.

Memory governor

MSE's memory governor feature solves the problem with using the stevedore to control the memory usage of Varnish. Instead of assigning a fixed amount of space for cache payloads, the memory governor aims to limit the total memory usage of the `varnishd` process, as seen by the operating system.

Instead limiting the size of the cache in *MSE*, by setting the `memcache_size` configuration directive to an absolute value, we can set it to `auto` to limit the total memory usage of the `varnishd` process instead.

When the memory governor observes that the memory usage is too high, it will start removing objects from the cache until the memory usage goes under the limit. This means that the actual memory used by *object payloads* will vary when other memory usage varies, but the total will be near constant. For example, if there are suddenly thousands of connections coming in, and thousands of threads need to be started to serve the connections, the extra memory usage from the connection handling will result in some objects being removed from memory until things calm down. If MSE with persistence is in use, no objects will be removed from the cache, just from the memory part of the cache.

The `memory_target` runtime parameter, which is set at *80%* by default, will ensure `varnishd` remains below that memory consumption ratio. `memory_target` can also be set to an absolute value, just like you would with the `-s` parameter.

When you set `memory_target` to a negative value, you can indicate how much memory you want the *memory governor* to leave unused.

The `memory_target` can also be changed at runtime. This can be useful if you need some memory for a different process and need `varnishd` to use less memory for a while.

The `memory_target` does not include the part of the kernel's *page cache* that is used to keep frequently used parts of the book in memory for fast access. For this reason, it might be necessary to tune down the `memory_target` parameter if your cache contains a lot of objects, and the book usage, measured in bytes, starts to creep up towards 10% of your available memory. It is not necessarily bad to have *some* paging activity as long as it stays under control.

When running *MSE*, it is a good idea to monitor paging behavior on the system, for example by using the `vmstat` tool. If it suddenly goes through the roof, you should consider reducing `memory_target` and see if it helps. Remember that this can be done on a running *Varnish* without restarting the service.

Debt collection

Enforcing the *memory target* is done similarly to the *waterlevel & hysteresis* mechanisms inside the persistence layer: it is also an *over/under* measurement.

When `varnishd` requests memory from the *OS* that results in exceeding the `memory_target`, debt is collected, which should quickly be repaid.

Repaying debt is done by removing objects from the cache on an *LRU* basis. Repaying accumulated debt is a shared responsibility:

- Fetches that contribute to the debt should repay it themselves.
- General debt is repaid by the *debt collector thread*.

If a fetch needs to store a *2 MB* object, and as a consequence surpasses `memory_target`, it needs to remove *2 MB* of content from the cache using *LRU*.

The *debt collector thread* will ensure `varnishd`'s memory consumption goes below the `memory_target` by removing objects from cache until the target is reached.

Funnily enough, the *debt collector thread* is nicknamed *the governator* because in order to govern, it needs to terminate objects.

Lucky loser

Varnish Cache suffers from a concept called *the lucky loser effect*.

When a fetch in *Varnish Cache* needs to free up space from the cache in order to facilitate its cache insert, it risks losing that space to a competing fetch.

That competing fetch was also looking for free space and happened to find it because the other fetch freed it up. This one is the lucky loser, but it results in a retry from the original fetch.

In theory the fetch can get stuck in a loop, continuously trying to free up space but failing to claim it. The `nuke_limit` runtime parameter defines how many times a fetch may try to free up space. The standard value is 50.

This concept can become detrimental to the performance of *Varnish*. The originating request will be left waiting until the object is stored in cache or until `nuke_limit` forces `varnishd` to bail out.

Luckily *Varnish Enterprise* doesn't suffer from this limitation when *MSE* is used. The standard *MSE* implementation has a level of isolation such that other fetches cannot see space that was freed up by other fetches.

When the *memory governor* is enabled, every fetch can only allocate objects to memory if they can cover the debt.

Basically, the unfairness is gone, which benefits performance.

7.4.3 Configuration

Enabling *MSE* is pretty simple. It's just a matter of adding `-s mse` to the `varnishd` command, and you're good to go. This will give you *MSE* in *memory-only* mode with the *memory governor* enabled. However, in most cases, you'll want to specify a bit more configuration.

As mentioned earlier in this chapter, you can point your storage configuration to an *MSE configuration file*. Here's a typical example:

```
varnishd -s mse,/var/lib/mse/mse.conf
```

The `/var/lib/mse/mse.conf` contains both the *memory-caching* configuration, and the *cache-persistence* configuration.

Memory configuration

You can set the size of the *memcache* in the *MSE config file*, and it will have the same effect as specifying the size of a `malloc` stevedore.

Here's an example:

```
  env: {
     id = "mse";
     memcache_size = "5G";
 };
```

This *MSE* configuration is named `mse` and allocates *5 GB* for object caching. This configuration will solve the *lucky loser* problem described above but will otherwise be equivalent to a `malloc` *stevedore* set to *5 GB*.

You can add some more configuration parameters to the environment. Here's an example:

```
env: {
    id = "mse";
    memcache_size = "5G";
    memcache_chunksize = "4M"
    memcache_metachunksize = "4K"
};
```

These two extra parameters define the maximum memory chunk size that can be allocated. One is for objects in general, and the second is an indication of the size of the metadata for such an object.

As previously mentioned, omitting the configuration file in the `varnishd` command line will enable the *memory governor*. If you do specify a configuration file, the *memory governor* can be enabled by setting `memcache_size` to `"auto"`, as illustrated below:

```
env: {
    id = "mse";
    memcache_size = "auto";
};
```

Persistence

Although *MSE* is a really good *memory cache*, most people will enable *persistence*.

While *persistence* is an important *MSE feature*, most people just want to cache more objects than they can fit in memory. Either way, you need *books* and *stores*.

Here's a simple example that was already featured earlier in this chapter:

```
env: {
    id = "mse";
    memcache_size = "auto";

    books = ( {
        id = "book";
        directory = "/var/lib/mse/book";
        database_size = "2G";

        stores = ( {
            id = "store";
            filename = "/var/lib/mse/store.dat";
            size = "100G";
        } );
    } );
};
```

This example uses a single book, which is located in `/var/lib/mse/book`, and a single store, located in `/var/lib/mse/store.dat`. The size of the *book* is limited to *2 GB*, and the *store* to *100 GB*. Meanwhile the *memory governor* is enabled to automatically manage the size of the memory cache.

`varnishd` will not create the files that are required for *persistence* to work. You have to initialize those paths yourself. *Varnish Enterprise* ships with an `mkfs.mse` program that reads the configuration file and creates the necessary files.

The following example uses `mkfs.mse` to create the necessary files, based on the `/var/lib/mse/mse.conf` configuration file:

```
$ sudo mkfs.mse -c /var/lib/mse/mse.conf
Creating environment ‘mse’
Creating book ‘mse.book’ in ‘/var/lib/mse/book’
Creating store ‘mse.book.store’ in ‘/var/lib/mse/store.dat’
Book ‘mse.book’ created successfully
Environment ‘mse’ created successfully
```

It is also possible to configure multiple stores:

```
env: {
    id = "mse";
    memcache_size = "auto";

    books = ( {
        id = "book";
        directory = "/var/lib/mse/book";
        database_size = "2G";

        stores = ( {
            id = "store1";
            filename = "/var/lib/mse/store1.dat";
            size = "100G";
        },{
            id = "store2";
            filename = "/var/lib/mse/store2.dat";
            size = "100G";
        } );
    } );
};
```

The individual store files can be stored on multiple disks to reduce risk, but also to benefit from the improved *I/O* performance. *MSE* will cycle through the *stores* using a *round-robin* algorithm.

It is also possible to have multiple *books*, each with their own *stores*:

```
env: {
    id = "mse";
    memcache_size = "auto";

    books = ( {
        id = "book1";
        directory = "/var/lib/mse/book1";
        database_size = "2G";

        stores = ( {
            id = "store1";
            filename = "/var/lib/mse/store1.dat";
            size = "100G";
        },{
            id = "store2";
            filename = "/var/lib/mse/store2.dat";
            size = "100G";
        } );
    },{
        id = "book2";
        directory = "/var/lib/mse/book2";
        database_size = "2G";

        stores = ( {
            id = "store3";
            filename = "/var/lib/mse/store3.dat";
            size = "100G";
        },{
            id = "store4";
            filename = "/var/lib/mse/store4.dat";
            size = "100G";
        } );
    } );
};
```

In this case the *metadata databases* are in different locations. It would make sense to host them on separate disks as well, just like the stores.

Although *books* and their *stores* form a unit, *MSE* will, when using its *round-robin* algorithm to find somewhere to store the object, only cycle through the list of *stores*, ignoring their relationship with *books*.

Book configuration

Books have various configuration directives, some of which have already been discussed. Although the default values will do for most people, it is worth noting that changing can be impactful, depending on your use case.

`id` is a required parameter and is used to name the book. `directory` is also required, as it refers to the location where the *LMDB database*, lockfiles, and journals that comprise the *book* will be hosted.

Here's a list of parameters that can be tuned:

- `database_size`: the total size of the *LMDB database* of the *book*. Defaults to *1 GB*
- `database_readers`: the maximum number of simultaneous database readers. Defaults to *4096*
- `database_sync`: whether or not to wait until the disk has confirmed the disk write of a change in the *LMDB database*. Defaults to *true*, which ensures data consistency. Setting it to *false* will increase performance, but you risk data corruption when a server outage occurs before the latest changes are synchronized to the disk.
- `database_waterlevel`: the maximum fill level of the *LMDB database*. Defaults to *0.9*, which is *90%*
- `database_waterlevel_hysterisis`: the *over/under ratio* to maintain when enforcing the *waterlevel* of the *LMDB database*. Defaults to *0.05*, which is a *5% over/under* on the *90%* that was defined in `database_waterlevel`
- `database_waterlevel_snipecount`: the number of objects to remove in one batch when enforcing the *waterlevel* for the database. Defaults to *10*
- `banlist_size`: the size of the *ban list journal*. Defaults to *1 MB*. Exceeding this limit will cause new bans to overflow into the *LMDB database*.

Store configuration

Similar configuration parameters are available for tuning *stores*. It all starts with two required parameters:

- `id`: the unique identifier of a store
- `filename`: the location on disk of the store file

Unlike a *book*, which is a collection of files in a directory, a *store* consists of a single file. The size of *store files* is defined by the `size` parameter, which defaults to *1 GB*.

These are the basic settings, but there are more configurable parameters. Here's a selection of configurable parameters:

- `align`: defaults to *4 KB* and defines store allocations to be multiples of this value
- `minfreechunk`: also defaults to *4 KB* and is the minimum size of a free chunk
- `aio_requests`: the number of simultaneous *asynchronous I/O* requests. Defaults to *128*
- `aio_db_handles`: the number of simultaneous *read-only handles* that are available for reading metadata from the corresponding *book*
- `journal_size`: the size of the journal that keeps track of incremental changes until they are applied to the corresponding *LMDB database*. Defaults to *1 M*
- `waterlevel_painted`: the fraction of objects that is painted as *LRU candidates* when the *waterlevel* is reached. By default this is *0.33*, which corresponds to *33%*
- `waterlevel_threads`: the number of threads that are responsible for enforcing the *waterlevel* and removing *LRU-painted objects* from the cache. Defaults to *1*
- `waterlevel_minchunksize`: the minimum chunk size that is considered when creating *continuous free space* when the *waterlevel* is exceeded. Defaults to *512 KB*
- `waterlevel`: the ratio of *continuous free space* that should be maintained. Defaults to *0.9* which corresponds to *90%*
- `waterlevel_hysterisis`: the *over/under ratio* to maintain when enforcing the *waterlevel* of the *store*. Defaults to *0.05*, which is a *5% over/under* on the *90%* that was defined in `waterlevel`
- `waterlevel_snipecount`: the number of objects to remove in one batch when enforcing the *waterlevel*. Defaults to *10*

The default settings for the *book* and *store* configuration have been carefully chosen by our engineers. We would advise you to stick with the default values unless you have specific concerns you want to address ahead of time, or if you're experiencing problems.

7.4.4 Store selection

Round-robin is the default way stores are selected in *MSE*; this can be changed in VCL through `vmod_mse`.

Stores can be selected individually by name, or you can select all the stores in a *book* by using the name of the *book*.

However, in most cases it is natural to apply *tags* to the *books* and/or *stores*, and use the *tags* to select the set of stores you want MSE to choose from. The selection of stores is a *core MSE feature*, but `vmod_mse` is used as an interface for this, when you need to override the default settings.

Tagging happens in the MSE configuration file, typically `/var/lib/mse/mse.conf`, and has not been discussed in this section up until this point.

Tagging stores

In your `/var/lib/mse/mse.conf` file you can use the `tags` directive to associate *tags* with individual *stores*.

Here's an example in which one store is hosted on a large *SATA disk*, and the other store is hosted on a smaller but much faster *SSD* disk:

```
env: {
    id = "mse";
    memcache_size = "auto";

    books = ( {
        id = "book";
        directory = "/var/lib/mse/book";

        stores = ( {
            id = "store1";
            filename = "/var/lib/mse/store1.dat";
            size = "500G";
            tags = ( "small", "ssd" );
        },{
            id = "store2";
            filename = "/var/lib/mse/store2.dat";
            size = "10T";
            tags = ( "big", "sata" );
        } );
    } );
};
```

In this case `store1` has *500 GB* of *SSD* storage at its disposal. That is at least what the tag indicates.

`store2` on the other hand is *10 TB* in size and has a `sata` tag linked to it. This would imply that this *store* has a larger but slower disk.

It is up to you to decide on naming of tags. Their names have no underlying significance. You can easily change the tag `ssd` into `fast`, and `sata` into `slow` if that is more intuitive to you.

These tags will be used in *VCL* when `vmod_mse` comes into play.

Tagging books

It is also possible to apply these tags on the *book* level. This means that all underlying *stores* will be tagged with these values.

Here's a *multi-book example*:

```
env: {
    id = "mse";
    memcache_size = "auto";

    books = ( {
        id = "book1";
        directory = "/var/lib/mse/book1";
        tags = ( "small", "ssd" );

        stores = ( {
            id = "store1";
            filename = "/var/lib/mse/store1.dat";
            size = "500G";
        },{
            id = "store2";
            filename = "/var/lib/mse/store2.dat";
            size = "500G";
        } );
    },{
        id = "book2";
        directory = "/var/lib/mse/book2";
        tags = ( "big", "sata" );

        stores = ( {
            id = "store3";
            filename = "/var/lib/mse/store3.dat";
            size = "10T";
        },{
            id = "store4";
            filename = "/var/lib/mse/store4.dat";
            size = "10T";
        } );
    } );
};
```

In this case `store1` and `store2` are tagged as `small` and `ssd` because these tags were applied to their corresponding *book*. They probably have smaller *SSD* disks in them, as the tags may imply.

For `store3` and `store4`, which are managed by `book2`, the tags are `big` and `sata`. No surprises here either: although we know about the size of the stores, which grant the `big` tag, we can only assume the underlying disks are *SATA disks*.

Simply adding tags, like in the example above, does not change anything. The default behavior, which is *round-robin* between all of the stores, still takes place until a set of stores is explicitly selected: either in VCL or in the configuration file.

Let's have a look at how this is done.

Setting the default stores

One way of selecting the default stores is by using the `default_stores` configuration directive in `/var/lib/mse/mse.conf`. This directive refers to a store based on its name, its tag, or the name or tag of the book.

Based on the example above we could configure the default stores as follows:

```
default_stores = "sata";
```

Unless instructed otherwise in *VCL*, the default stores will be `store3` and `store4`.

There is also a special value `none`, which does not refer to any tag.

```
default_stores = "none";
```

This example will ensure objects only get stored in memory cache unless instructed otherwise in *VCL*.

As soon as `default_stores` is set, the *round-robin store selection* no longer applies and is replaced by a *random* selection, where a potentially uneven weighting is applied based on the *store size*.

vmod_mse

You can have fine-grained control over your store selection, if you leverage `vmod_mse`.

This *VMOD* has an `mse.set_stores()` function that allows you to refer to a *book*, a *store*, or a *tag*. Also the special value `none` is allowed to bypass the persistence layer.

Here's an example where we select the *stores* based on *response-header criteria*:

```
vcl 4.1;

import mse;
import std;

sub vcl_backend_response {
    if (beresp.ttl < 120s) {
        mse.set_stores("none");
    } else {
        if (beresp.http.Transfer-Encoding ~ "chunked" ||
        std.integer(beresp.http.Content-Length,0) > std.bytes("100M"))
{
            mse.set_stores("sata");
        } else {
            mse.set_stores("ssd");
        }
    }
}
```

Let's break it down:

- Cache inserts with a *TTL* of less than *two minutes* will not be persisted and will only be cached in memory.
- Cache inserts where the `Content-Length` response header indicates a size of more than *100 MB* will be stored on the *sata-tagged stores*.
- Cache inserts where the `Transfer-Encoding` response header is set to `chunked` also end up on the *sata-tagged stores*. Because data is streamed to *Varnish*, we have no clue of the size ahead of time.
- All other cache inserts are less than *100 MB* in size and will end up on the *ssd-tagged stores*.

When one or more stores are selected based on a tag or name, either in VCL or by using `default_stores`, the actual destination of the object will always be determined by a fast *quasi random number generator*. This means that if you add just a few objects to your MSE, you should expect the distribution to be uneven, but for any reasonable number of objects, the unevenness should be negligible.

You can change the weighting of stores through the function `mse.set_weighting()` to one of the following:

- `size`: bigger *stores* have a higher probability of being selected.
- `available`: *stores* with more available space have a higher probability of being selected.
- `smooth`: *store* size and availability of space are combined to assign weights to stores.

As you already know, `size` becomes the default weighting mechanism when a store is selected through `default_stores` or `mse.set_stores()`. The `mse.set_weighting()` function also allows you to set weighting mechanisms.

Here's an example of how to set the weighting to `smooth:`:

```
vcl 4.1;

import mse;

sub vcl_backend_response {
    mse.set_weighting(smooth);
}
```

7.4.5 Monitoring

Although there is a dedicated section about *monitoring* coming up later in this chapter, we do want to hint at monitoring internal *MSE* counters using `varnishstat`.

Memory counters

Here's a table with some counters that relate to *memory caching*:

Counter	Meaning
`MSE.mse.g_bytes`	Bytes outstanding
`MSE.mse.g_space`	Bytes available
`MSE.mse.n_lru_nuked`	Number of LRU-nuked objects
`MSE.mse.n_vary`	Number of Vary header keys
`MSE.mse.c_memcache_hit`	Stored objects cache hits
`MSE.mse.c_memcache_miss`	Stored objects cache misses
`MSE.mse.g_ykey_keys`	Number of YKeys registered
`MSE.mse.c_ykey_purged`	Number of objects purged with YKey

If you run the following command, you can use the `MSE.mse.g_space` counter to see how much space is left in memory for caching:

```
varnishstat -f MSE.mse.g_space
```

The `mse` keyword in these counters refers to the name of your environment. In this case it is named `mse`. If you were to name your environment `server1`, the counter would be `MSE.server1.g_space`. If you want to make sure you see all environments, you can use an asterisk, as illustrated below:

```
varnishstat -f MSE.*.g_space
```

Book counters

There are also counters to monitor the state of your *books*. Here's a table with some select counters related to *books*:

Counter	Meaning
MSE_BOOK.book1.n_vary	Number of Vary header keys
MSE_BOOK.book1.g_bytes	Number of bytes used in the book database
MSE_BOOK.book1.g_space	Number of bytes available in the book database
MSE_BOOK.book1.g_waterlevel_queue	Number of threads queued waiting for database space
MSE_BOOK.book1.c_waterlevel_queue	Number of times a thread has been queued waiting for database space
MSE_BOOK.book1.c_waterlevel_runs	Number of times the waterlevel purge thread was activated
MSE_BOOK.book1.c_waterlevel_purge	Number of objects purged to achieve database waterlevel
MSE_BOOK.book1.c_insert_timeout	Number of times database object insertion timed out
MSE_BOOK.book1.g_banlist_bytes	Number of bytes used from the ban list journal file

`MSE_BOOK.book1.g_banlist_space`	Number of bytes available in the ban list journal file
`MSE_BOOK.book1.g_banlist_database`	Number of bytes used in the database for persisted bans

These counters specifically refer to `book1`, but as there are multiple *books*, it makes sense to query on all *books*, as illustrated below:

```
varnishstat -f MSE_BOOK.*
```

This command shows all counters for all *books*.

Store counters

And finally, *stores* also have their own counters. Here's the table:

Counter	Meaning
`MSE_STORE.store1.g_waterlevel_queue`	Number of threads queued waiting for store space
`MSE_STORE.store1.c_waterlevel_queue`	Number of times a thread has been queued waiting for store space
`MSE_STORE.store1.c_waterlevel_purge`	Number of objects purged to achieve store waterlevel
`MSE_STORE.store1.g_objects`	Number of objects in the store
`MSE_STORE.store1.g_ykey_keys`	Number of YKeys registered
`MSE_STORE.store1.c_ykey_purged`	Number of objects purged with YKey
`MSE_STORE.store1.g_alloc_bytes`	Total number of bytes in allocation extents
`MSE_STORE.store1.g_free_bytes`	Total number of bytes in free extents
`MSE_STORE.store1.g_free_small_bytes`	Number of bytes in free extents smaller than 16k
`MSE_STORE.store1.g_free_16k_bytes`	Number of bytes in free extents between 16k and 32k
`MSE_STORE.store1.g_free_32k_bytes`	Number of bytes in free extents between 32k and 64k

`MSE_STORE.store1.g_free_64k_bytes`	Number of bytes in free extents between 64k and 128k
`MSE_STORE.store1.g_free_128k_bytes`	Number of bytes in free extents between 128k and 256k
`MSE_STORE.store1.g_free_256k_bytes`	Number of bytes in free extents between 256k and 512k
`MSE_STORE.store1.g_free_512k_bytes`	Number of bytes in free extents between 512k and 1m
`MSE_STORE.store1.g_free_1m_bytes`	Number of bytes in free extents between 1m and 2m
`MSE_STORE.store1.g_free_2m_bytes`	Number of bytes in free extents between 2m and 4m
`MSE_STORE.store1.g_free_4m_bytes`	Number of bytes in free extents between 4m and 8m
`MSE_STORE.store1.g_free_large_bytes`	Number of bytes in free extents larger than 8m

Besides the typical free space, bytes allocated, and number of objects in the *store*, you'll also find detailed counters on the *extents per size*. This is part of the *anti-fragmentation mechanism* that aims to have *continuous free space* but tolerates a level of fragmentation below the `waterlevel_minchunksize`.

In the beginning, there will be no fragmentation within your store, so the `MSE_STORE.store1.g_free_large_bytes` counter will be high, the others will be low or zero.

The following command will monitor the *free bytes per extent* for all *stores*:

```
varnishstat  -f MSE_STORE.*.g_free_*
```

A more basic situation to monitor is the number of objects in cache, the available space in the *stores*, and the used space. Here's how you do that:

```
varnishstat -f MSE_STORE.*.g_objects -f MSE_STORE.*.g_free_bytes -f
MSE_STORE.*.g_alloc_bytes
```

7.4.6 Cache warming

The promise of *MSE* is persistence. You benefit from this when a server restart occurs: *stores* and *books* will be loaded from disk and the full context is restored.

If you take a backup of your *stores* and *books*, you can restore this backup and perform a *disaster recovery*.

Even when your *books* and *stores* are corrupted, or even gone, the backup will restore the previous state, and your cache will be *warm*.

Besides *disaster recovery*, this strategy can also be used to *pre-warm* the cache on new *Varnish* instances.

When you restart `varnishd`, and the recovered *books* and *stores* are found, the output can be contain the following:

```
varnish    | Info: Child (24) said Store mse.book1.store1 revived 6
objects
varnish    | Info: Child (24) said Store mse.book1.store1 removed 0
objects (partial=0 age=0 marked=0 noban=0 novary=0)
varnish    | Info: Child (24) said Store mse.book1.store2 revived 2
objects
varnish    | Info: Child (24) said Store mse.book1.store2 removed 0
objects (partial=0 age=0 marked=0 noban=0 novary=0)
varnish    | Info: Child (24) said Store mse.book2.store3 revived 6
objects
varnish    | Info: Child (24) said Store mse.book2.store3 removed 0
objects (partial=0 age=0 marked=0 noban=0 novary=0)
varnish    | Info: Child (24) said Store mse.book2.store4 revived 3
objects
varnish    | Info: Child (24) said Store mse.book2.store4 removed 0
objects (partial=0 age=0 marked=0 noban=0 novary=0)
varnish    | Info: Child (24) said Environment mse fully populated in
0.00 seconds. (0.00 0.00 0.00 17 0 3/4 4 0 4 0)
```

As you can see, objects were revived. In huge caches with lots of objects, loading the environment might take a bit longer.

MSE can easily load more than a million objects per second. Although the time it takes for *MSE* to load objects from disk depends on the kind of hardware you use, we can assume that for bigger caches this would only take mere seconds.

7.5 Load balancing

When a *Varnish* server is tasked with proxying backend requests to multiple origin servers, it is important that the right backend is selected.

If there is an affinity between the client and a specific backend, or the request and a specific backend, *VCL* offers you the flexibility to define rules and add the required logic on which these backend routing decisions are based.

VCL has the `req.backend_hint` and the `bereq.backend` variables that can be set to assign a backend. This allows you to make backend routing decisions based on *HTTP request or client information*.

There are also situations where you don't want to select one backend, but you want all backends from the pool to participate. The goal is to distribute requests across those backends for scalability reasons. We call this *load balancing*.

Varnish has a *VMOD* called `vmod_directors`, which takes care of load balancing. This *VMOD* can register backends, and based on a distribution algorithm, a backend is selected on a per-request basis.

Even though load balancing aims to evenly distribute the load across all servers in the pool, some directors allow you to configure a level of affinity with one or more backends.

7.5.1 Directors

The *directors VMOD* is an in-tree *VMOD* that is shipped with *Varnish* by default. It has a relatively consistent *API* for initialization and for adding backends.

A *director object* will pick a backend when the `.backend()` method is called. The selected backend can be assigned to *Varnish* using `req.backend_hint` and `bereq.backend`.

7.5.1 Round-robin director

The *round-robin director* will create a *director object* that will cycle through backends every time `.backend()` is selected.

Here's a basic *round-robin* example with three backends:

```
vcl 4.1;

import directors;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

backend backend3 {
    .host = "backend3.example.com";
    .port = "80";
}

sub vcl_init {
    new vdir = directors.round_robin();
    vdir.add_backend(backend1);
    vdir.add_backend(backend2);
    vdir.add_backend(backend3);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend();
}
```

`new vdir = directors.round_robin()` will initialize the *round-robin director object*. The `vdir.add_backend()` method will add the three backends to the *director*.

And every time `vdir.backend()` is called, the *director* will cycle through those backends.

Because it is done in a *round-robin* fashion, the order of execution is very predictable.

The output below comes from the `varnishlog` binary that filters on the `BackendOpen` tag that indicates which backend is used:

```
$ varnishlog -g raw -i BackendOpen
     32786 BackendOpen    b 26 boot.backend1 172.21.0.2 80 172.21.0.5
56422
        24 BackendOpen    b 27 boot.backend2 172.21.0.4 80 172.21.0.5
45792
     32789 BackendOpen    b 28 boot.backend3 172.21.0.3 80 172.21.0.5
54702
        27 BackendOpen    b 26 boot.backend1 172.21.0.2 80 172.21.0.5
56422
     32792 BackendOpen    b 27 boot.backend2 172.21.0.4 80 172.21.0.5
45792
     32795 BackendOpen    b 28 boot.backend3 172.21.0.3 80 172.21.0.5
54702
```

As you can see `backend1` is used first, then `backend2`, and finally `backend3`. This order of execution is respected for subsequent requests.

Round-robin will ensure an equal distribution of load across all origin servers.

7.5.1 Random director

The *random director* will distribute the load using a weighted random probability distribution.

The *API* doesn't differ much from the *round-robin director*. In the snippet below, there is *equal weighting*:

```
sub vcl_init {
    new vdir = directors.random();
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 1);
    vdir.add_backend(backend3, 1);
}
```

The formula that is used to determine the weighting is `100 * (weight / .(sum(all_added_weights)))`.

Here's another *VCL snippet* with unequal weighting:

```
sub vcl_init {
    new vdir = directors.random();
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 2);
    vdir.add_backend(backend3, 3);
}
```

If we apply the formula for this example, the distribution is as follows:

- `backend1` has a *16.66%* probability of being selected.
- `backend2` has a *33.33%* probability of being selected
- `backend3` has a *50%* probability of being selected.

These weights are useful when some of the backends shouldn't receive the same amount of traffic. This may be because they don't have the same dimensions and are less powerful.

Watch out: setting the *weight* of a backend to *zero* gives the backend a *zero percent probability* of being selected.

7.5.1 Fallback director

Another kind of *load balancing* we can use in *Varnish* is only based on potential failure.

A *fallback director* will try each of the added backends in turn and return the first one that is healthy.

Configuring this type of *director* is very similar to the *round-robin* one. Here's the `vcl_init` snippet:

```
sub vcl_init {
    new vdir = directors.fallback();
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 1);
    vdir.add_backend(backend3, 1);
}
```

- `backend1` is the main backend, and will always be used if it is healthy.
- When `backend1` fails, `backend2` becomes the selected backend.
- And if both `backend1` and `backend2` fail, `backend3` is used.

If a higher-priority backend becomes healthy again, it will become the main backend.

By setting the *sticky* argument to `true`, the *fallback director* will stick with the selected backend, even if a higher-priority backend becomes available again.

Here's how you enable *stickiness*:

```
sub vcl_init {
    new vdir = directors.fallback(true);
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 1);
    vdir.add_backend(backend3, 1);
}
```

7.5.1 Hash director

The *hash director* is used to consistently send requests to the same backend, based on a *hash* that is computed by the *director* and that is associated with a backend.

The example below contains a very common use case: *sticky IP*. This means that requests from a client are always sent to the same backend.

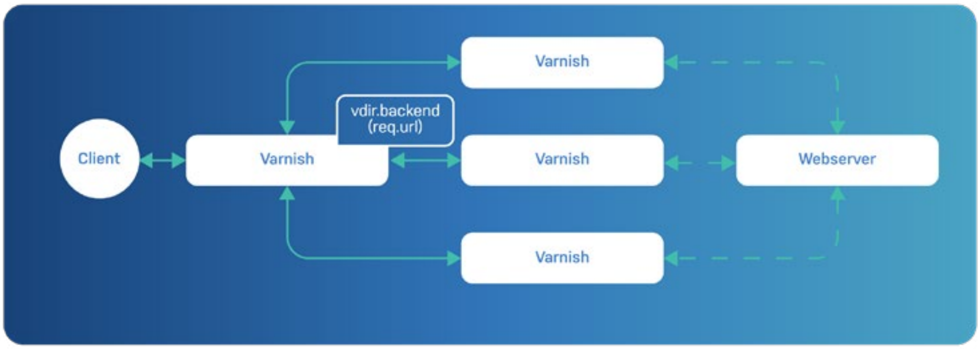
```
vcl 4.1;

import directors;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

backend backend3 {
    .host = "backend3.example.com";
    .port = "80";
}

sub vcl_init {
    new vdir = directors.hash();
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 1);
    vdir.add_backend(backend3, 1);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend(client.ip);
}
```

Routing through two layers of Varnish

If you want to horizontally scale your cache, you can use the *hash director* to send all requests for the same *URL* to the same *Varnish* server. To achieve this, you need two layers of *Varnish*:

- The *routing layer* that performs the hashing
- The *caching layer* that stores the objects in cache

The following diagram illustrates this:



Hash director

The top-level *Varnish* server, which acts as a *router*, can use the following *VCL code* to evenly distribute the content across to lower-level *Varnish* servers:

```
vcl 4.1;

import directors;

backend varnish1 {
    .host = "varnish1.example.com";
    .port = "80";
}

backend varnish2 {
    .host = "varnish2.example.com";
    .port = "80";
}

backend varnish3 {
    .host = "varnish3.example.com";
    .port = "80";
}
```

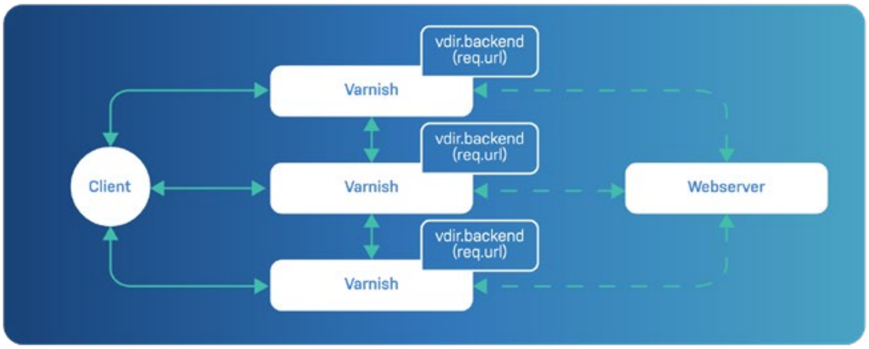
```
sub vcl_init {
    new vdir = directors.hash();
    vdir.add_backend(varnish1, 1);
    vdir.add_backend(varnish2, 1);
    vdir.add_backend(varnish3, 1);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend(req.url);
}
```

By scaling horizontally, you can cache a lot more data than on a single server. The *hash director* ensures there is not content duplication on the lower-level nodes. And if required, the top-level *Varnish* server can also cache some of the *hot content*.

Self-routing Varnish cluster

Whereas the previous example was quite *vertical*, the next one has the same capabilities but structured *horizontally*.

Imagine the setup featured in this diagram:



Self-routing Varnish cluster

What you have is three *Varnish* servers that are aware of each other. The `vdir.backend(req.url)` method creates a hash and selects a node.

When *Varnish* notices that the selected node has the same IP address as it has, it routes the request to the origin server. If the IP address is not the same, the request is routed to another *Varnish* node.

What is also interesting to note is that all *Varnish* servers create the same hash, so the outcome is predictable.

Here's the *VCL* code:

```
vcl 4.1;

import directors;

backend varnish1 {
    .host = "varnish1.example.com";
    .port = "80";
}

backend varnish2 {
    .host = "varnish2.example.com";
    .port = "80";
}

backend varnish3 {
    .host = "varnish3.example.com";
    .port = "80";
}

backend origin {
  .host = "origin.example.com";
  .port = "80";
}

sub vcl_init {
    new vdir = directors.hash();
    vdir.add_backend(varnish1, 1);
    vdir.add_backend(varnish2, 1);
    vdir.add_backend(varnish3, 1);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend(req.url);
    set req.http.x-shard = req.backend_hint;
    if (req.http.x-shard == server.identity) {
        set req.backend_hint = origin;
    } else {
        return(pass);
    }
}
```

Key remapping

The *hash director* is a relatively simple and powerful implementation, but when nodes are added or temporarily removed, a lot of keys have to be remapped.

Although there is a level *consistency*, the *hash director* doesn't apply a true *consistent hashing* algorithm.

When a backend that is part of your *hash director* is taken out of commission, not only the hashes that belonged to that server have to be remapped to the remaining nodes, a lot of other hashes from healthy servers do as well.

This is more or less done by design, as the main priority of the *hash director* is fairness: keys have to be equally distributed across the backend servers to avoid overloading a single backend.

7.5.1 Shard director

The *shard director* behaves very similarly to the *hash director*: a hash is composed from a specific key, and this hash is consistently mapped to a backend server.

However, the *shard director* has a lot more options it can configure. It also applies a real *consistent hashing* algorithm with replicas, which we'll talk about in a minute.

Its main advantage is that when the backend configuration or health state changes, the association of keys to backends remains as stable as possible.

In addition, the ramp-up and warmup features can help to further improve user-perceived response times.

Here's an initial *VCL example* where the *request hash* from `vcl_hash` is used as the key. This hash is consistently mapped to one of the backend servers:

```
vcl 4.1;

import directors;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

backend backend3 {
    .host = "backend3.example.com";
    .port = "80";
}
```

```
sub vcl_init {
    new vdir = directors.shard();
    vdir.add_backend(backend1);
    vdir.add_backend(backend2);
    vdir.add_backend(backend3);
    vdir.reconfigure();
}

sub vcl_backend_fetch {
    set bereq.backend = vdir.backend();
}
```

Hash selection

The *shard director* can also pick an arbitrary key to hash. Although the `.backend()` method doesn't need any input parameters, it does default to `HASH`. As explained earlier, this is the request hash from `vcl_hash`.

You can also hash in the *URL*, which differs from the complete hash because the *hostname* and any custom variations will be missing.

Here's how you configure *URL hashing*:

```
sub vcl_backend_fetch {
    set bereq.backend = vdir.backend(URL);
}
```

Just like the *hash director*, you can hash an arbitrary key. If we want to create *sticky sessions* and use the *client IP address* to consistently route clients to the same backend, we can use the following snippet:

```
sub vcl_backend_fetch {
    set bereq.backend = vdir.backend(KEY,vdir.key(client.ip));
}
```

Warmup and ramp-up

The *shard director* has a *warmup* and a *ramp-up* feature. Both are related to gradually introducing traffic to a backend, though *warmup* is about gradually sending requests to other backends, and *ramp-up* is about gradually reintroducing the main backend.

The first snippet will set the *default warmup probability* to `0.5`:

```
vdir.set_warmup(0.5)
```

This method will set the warmup on all backends and ensures that around *50%* of all requests will be sent to an alternate backend. This is done to *warm up* that node in case it gets selected.

Warmup only works on healthy nodes, can only happen if a node is not in *ramp-up*, and if the *alternate backend selection* didn't explicitly happen in the `.backend()` method.

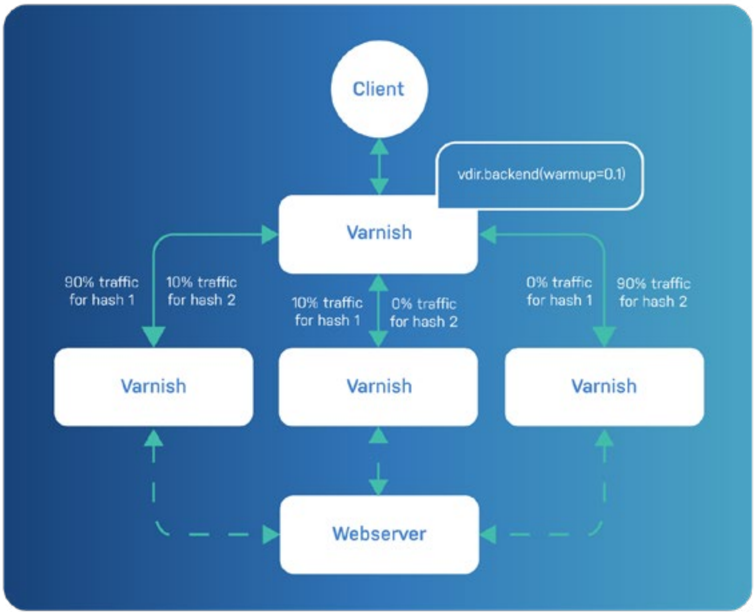
Here's an example where the *warmup* value is set upon backend selection:

```
sub vcl_backend_fetch {
    set bereq.backend = vdir.backend(by=URL, warmup=0.1);
}
```

In this case the *warmup value* will send about *10%* of traffic to the alternate backend, and it also uses the *URL* for hashing.

Warming up an alternate backend doesn't seem that useful when you talk about regular web servers, but if you look at it from a two-layer *Varnish* setup, it definitely make sense.

Here's an illustration where *warmup* is used to make sure second-tier *Varnish* servers are warmed up in case other nodes fail:



Shard director warmup

Whereas *warmup* happens on healthy servers, *ramp-up* happens on servers that have recently become healthy, either because they are new or because they recovered from an outage.

Ramp-up can be set globally, or on a per-backend basis. Here's a *VCL snippet* that sets the global ramp-up to a minute:

```
vdir.set_rampup(1m);
```

When a backend becomes healthy again, the relative weight of the backend is pushed all the way down, and gradually increases for the duration of the ramp-up period.

While a backend is ramping up, it receives a fraction of its normal traffic, while the next alternative backend takes the rest. Eventually this smooths out, and after a while the backend can be considered fully operational.

Ramp-up can only happen when the alternative backend server was not explicitly set in the `.backend()` method.

Here's a snippet where the *ramp-up period* differs per backend:

```
sub vcl_init {
    new vdir = directors.shard();
    vdir.add_backend(backend1, rampup=5m);
    vdir.add_backend(backend2, rampup=30s);
    vdir.add_backend(backend3);
    vdir.reconfigure();
}
```

In this case, `backend1` has a *five-minute rampup period*, whereas `backend2` has a *ten-second rampup period*. `backend3`, however, takes its *rampup duration* from the global setting.

When `.backend()` is executed and a backend is selected, *rampup* is enabled by default unless *rampup durations* are set to `0s`.

It is possible to still disable *rampup* on a per-backend request basis:

```
sub vcl_backend_fetch {
    set bereq.backend = vdir.backend(rampup=false);
}
```

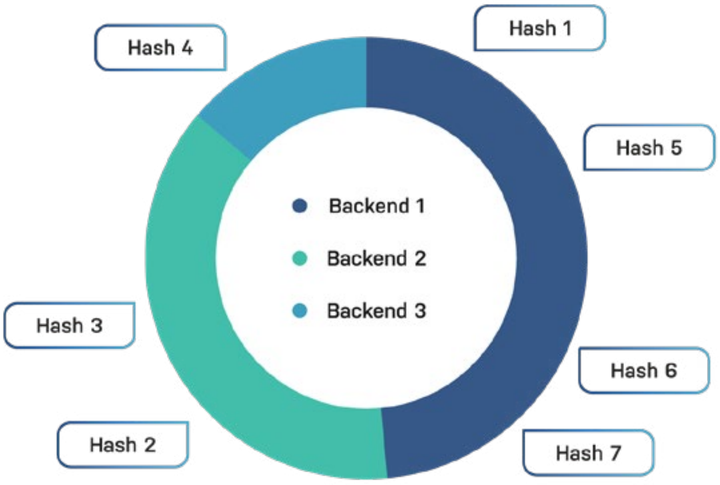
Key mapping and remapping

What makes the *shard director* so interesting is the fact that it uses *consistent hashing* with replica support.

Imagine the *shard director* as ring where each backend covers parts of the ring. Not every backend gets an equal amount of space. This is decided somewhat randomly.

Hashes are assigned to specific backends and because equidistribution is not a priority, some backends may receive a disproportionate number of requests.

Here's a simplistic pie chart that illustrates this concept:



Consistent hashing with a single replica

In this case, *backend 3* was unlucky, and only has a single hash mapped to it, whereas the other backends each have at least two hashes. This example only uses a single replica.

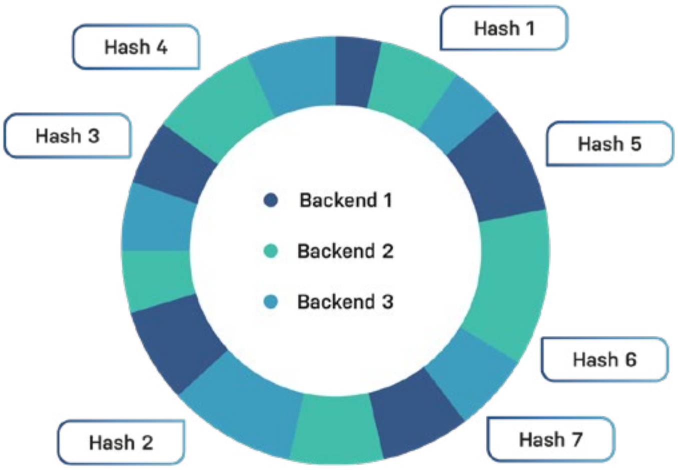
When the `.backend()` method is called, the smallest hash value larger than the hash itself is looked up on the circle. The movement is clockwise and may wrap around the circle. The backend that has the corresponding hash on its surface is selected.

When a backend is out of commission, its keys are remapped to other nodes while it is unavailable. When the backend becomes healthy again, it will receive its original hashes again.

Because a level of randomness is introduced, certain backends may be linked to a lot of hashes, whereas other backends aren't. This results in a higher load on a single backend and less fairness. In the chart above, *backend 3* has four out of seven hashes it takes care of.

By increasing the number of *replicas* per backend, every backend has more occurrences on the ring, which results in a fairer distribution. The default value for the *shard director* is *67*.

But for the sake of simplicity, here's an example with five replicas:



Consistent hashing with 5 replicas

As you can see the distribution is a lot fairer. Of course this doesn't matter for seven hashes, but as the hash count increases, the effect of replication does as well.

A simple simulation with three backends using a single replica and 100 total requests yielded the following results:

- *Backend 1* received *484* requests.
- *Backend 2* received *363* requests.
- *Backend 3* received *153* requests.

This ratio is represented in the first pie chart.

When we increased the replica count to *67*, the following results came back:

- *Backend 1* received *301* requests.
- *Backend 2* received *405* requests.
- *Backend 3* received *294* requests.

This is somewhat better and represents the ratios in the second pie chart. When we increased the replica count to *250*, the distribution was even more equal:

- *Backend 1* received *339* requests.
- *Backend 2* received *320* requests.
- *Backend 3* received *341* requests.

Although equidistribution is nice, it comes at a cost. The more replicas you define, the higher the *CPU cost* with diminishing returns.

You can configure the *replica count* in the `.reconfigure()` method.

Here's some example *VCL* that sets the replica count to 50:

```
sub vcl_init {
    new vdir = directors.shard();
    vdir.add_backend(backend1);
    vdir.add_backend(backend2);
    vdir.add_backend(backend3);
    vdir.reconfigure(50);
}
```

7.5.1 Least connections director

The *least connections director* is not part of `vmod_directors` but is a dedicated *VMOD* that is part of *Varnish Enterprise*.

It will route backend connections to the one with the least amount of connections at that point. An optional *ramp-up* configuration is also available.

Just like the *random director*, this one also uses weights to prioritize traffic to specific backends.

Here's some example *VCL* to illustrate how to use `vmod_leastconn`:

```
vcl 4.1;

import leastconn;

backend backend1 {
    .host = "backend1.example.com";
    .port = "80";
}

backend backend2 {
    .host = "backend2.example.com";
    .port = "80";
}

backend backend3 {
    .host = "backend3.example.com";
    .port = "80";
}

sub vcl_init {
    new vdir = leastconn.leastconn();
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 1);
    vdir.add_backend(backend3, 1);
}

sub vcl_recv {
    set req.backend_hint = vdir.backend();
}
```

And here's a *VCL snippet* where a *one-minute rampup* is used for backends that have become healthy:

```
sub vcl_init {
    new vdir = leastconn.leastconn();
    vdir.rampup(1m);
    vdir.add_backend(backend1, 1);
    vdir.add_backend(backend2, 1);
    vdir.add_backend(backend3, 1);
}
```

This means that when an unhealthy backend becomes healthy again, its weight is initially reduced, and gradually increases, until the configured weight is reached. In the example above, the weight increase happens over the course of a minute.

7.5.1 Dynamic backends

Remember `vmod_goto`, the *VMOD* that supports *dynamic backends*?

The `goto.dns.director()` function exposes a *director object* and fetches the associated IP addresses from the hostname. If multiple IP addresses are associated, *Varnish* cycles through them and performs *round-robin load balancing*.

Imagine having a pool of *origin servers* that is available via `origin.example.com` with the following IP addresses:

```
192.168.128.2
192.168.128.3
192.168.128.4
```

The *VCL example* below will extract these IP addresses via *DNS* and will perform *round-robin load balancing*:

```
vcl 4.1;

import goto;

backend default none;

sub vcl_init {
    new apipool = goto.dns_director("origin.example.com");
}

sub vcl_recv {
    set req.backend_hint = apipool.backend();
}
```

If you remember the strengths of `vmod_goto` from previous chapters, you'll understand that *DNS resolution* is not done at compile time but at runtime.

This means that if the hostname changes, `vmod_goto` will notice these changes and act accordingly. This way you can scale out your web server farm without having to reconfigure *Varnish*.

However, it is important to take *DNS TTLs* into account. A refresh of the hostname will only happen if the *TTL* has expired. DNS records have *TTLs*, and they can be quite high. You can also define a *TTL* in `goto.dns_director()`. Which one is considered?

The standard behavior is that `vmod_goto` will resolve the hostname every *ten seconds*. This can be overridden via the `ttl` argument.

You can also define a *TTL* rule in which you define to what extent the *TTL* from the DNS record is respected.

These are the possible values:

- `abide`: use the *TTL* that was extracted from the DNS record
- `force`: use the *TTL parameter* that was defined by `vmod_goto`
- `morethan`: use the *TTL* from the DNS record unless the *TTL parameter* is higher
- `lessthan`: use the *TTL* from the DNS record unless the *TTL parameter* is lower

Here's a *VCL snippet* where we enforce a *30-second TTL* unless the *TTL* extracted from the DNS record is less than 30 seconds:

```
sub vcl_init {
    new apipool = goto.dns_director("origin.example.com", ttl=30s,
ttl_rule=lessthan);
}
```

7.6 High Availability

Uptime, stability, performance, scalability. These are all operational priorities within a *content delivery* context. Nowadays, people have little tolerance for downtime, or even small hiccups in terms of stability.

Maintaining operational stability requires scaling out your infrastructure. As explained early on in the book: *Varnish* operates as an *origin shield*. It takes away a lot of load from the origin servers.

But even with *Varnish* protecting your origin layer, there is still a need to scale your infrastructure. You'll need at least two *Varnish* servers to ensure continuity if one of your *Varnish* servers were to go down.

This concept of taking precautions in case of individual failures is what we call *high availability*: you plan for failure, and you make sure fallback servers can take over when required.

A lot of *high availability* strategies have an *active-passive* setup, where only a part of the infrastructure is used at all times. In this case, a certain percentage of your infrastructure is only used when a failover needs to happen.

There's a level of inefficiency there, and a lot of companies are improving the inefficiency by opting to use *active-active* setups, where traffic is routed to all nodes.

An added benefit of scaling out for *high-availability* reasons is the fact that you're also scaling out for *performance* reasons: because you have a load-balanced setup with multiple *Varnish* nodes, you'll be able to handle a lot more traffic.

7.6.1 Keeping the caches hot

One of the challenges when dealing with multiple *Varnish* servers is keeping the caches hot. When a node fails, and the fallback server is *cold*, you'll start off with a lot of *cache misses*, which can have an immediate impact on the origin.

> A cache is *cold* when it has expired, is minimal or has no content.

Making sure your caches are synchronized avoids an additional strain on the origin. That is, if your load balancer does *round-robin* distribution, it is possible that a *hit* from the previous request now turns into a *miss* because the current node doesn't have the object in cache.

- On a two-node cluster this can lead to 50% more backend requests.
- On a three-node cluster this can lead to 66% more backend requests.
- On a four-node cluster this can lead to 75% more backend requests.

There are solutions out there where *Varnish* servers are chained to each other and act as each other's backend. There is some additional detection logic in there to avoid loops, but all in all this is sub-optimal, and not a real *HA solution*.

7.6.1 VHA

Varnish Enterprise comes with a full-blown *high-availability* suite called *Varnish High Availability*. But we'll just refer to it as *VHA*.

VHA will replicate cache inserts on one *Varnish* server to the other nodes in the cluster. The request that resulted in a *miss* and triggered the broadcast on the first node will now result in a cache *hit* when the equivalent request is received by the other servers in the cluster.

The *VHA* logic is written in *VCL*, requires a couple of *VMODs*, and primarily depends on the *Varnish Broadcaster* for replication.

7.6.1 Leveraging the broadcaster

A key aspect of *VHA* is performing the actual replication, and knowing which servers to send the data to. Instead of opting for a custom implementation, the *Varnish Broadcaster* was chosen as the replication mechanism.

The *broadcaster* is already an important tool in the *Varnish Enterprise* toolbox. As discussed in *chapter 6*, the *broadcaster* is commonly used to perform cache invalidations on multiple *Varnish* servers. In essence the *broadcaster*'s main role is in its name: broadcasting *HTTP messages*.

In *VHA*, the *broadcaster* will broadcast replication messages containing information about the inserted object. The exact details will be discussed when we talk about the architecture.

Another feature of the *broadcaster* is the `nodes.conf` file that contains the server inventory of the *Varnish cluster*. If you use the *broadcaster*, you're already using `nodes.conf`, which means you already defined the nodes in your cluster.

This concept can also be reused for *VHA*: it is very likely that the inventory that is used for cache invalidation will also be used for replication.

The *broadcaster* can either be hosted on your individual *Varnish* servers, or you can have a set of dedicated *broadcaster* servers that *Varnish* connects to.

The way *VHA* interacts with the *broadcaster* can be configured. We'll cover this later in this section about *VHA*.

7.6.1 Architecture

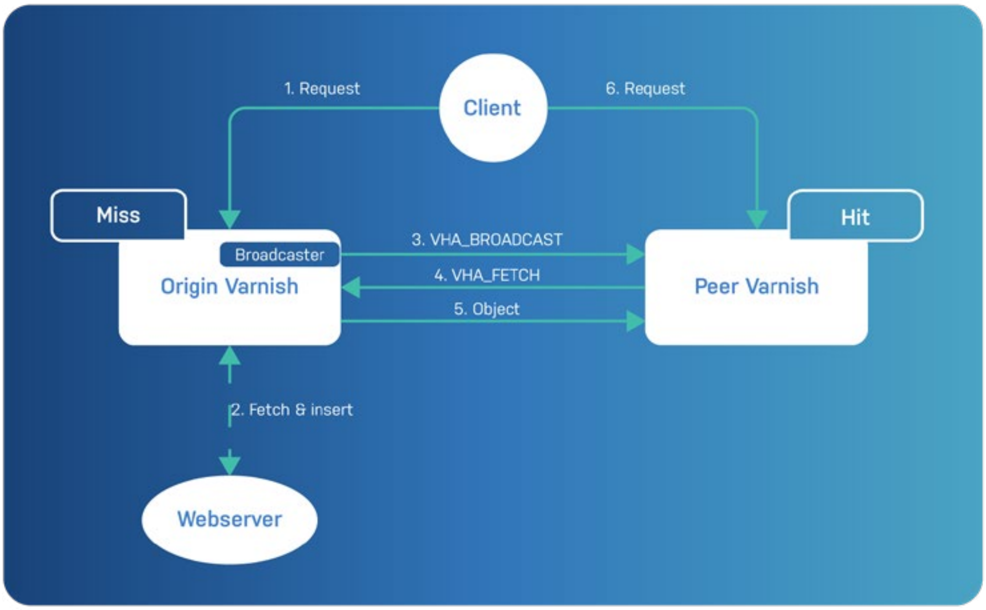
As mentioned, the *broadcaster* does a lot of the heavy lifting in *VHA*. The logic that decides what, when, and how broadcasting happens, is written in *VCL*.

The *Varnish* instance that initiates the replication is called the *VHA origin*. Yes, that may sound confusing because we have always called our backend servers the *origin*. But in the *VHA* context, the *origin* is the *Varnish* server that initiates the broadcast.

The server that receives the replication request is called the *VHA peer*.

Workflow

As you can see in the diagram below, *VHA* has a specific workflow:



VHA architecture

Let's go through the different steps:

36. A client sends an *HTTP request* to the first *Varnish* server.
37. The content is not in cache and will be fetched from the backend.
38. Upon cache insertion, the *VHA origin* will ask the *broadcaster* to send a `VHA_BROADCAST` request to the other nodes in the cluster.
39. The *VHA peers* that received this `VHA_BROADCAST` request will send a `VHA_FETCH` request to the *VHA origin* when they don't have the object in cache.
40. The *VHA origin* sends the object in the form of an *HTTP response* to the *VHA peer* that requested it. The *peer* then stores this object in cache for future requests.
41. When a client sends a request for that same resource to the second *Varnish* server, it will be able to serve it from cache.

Remember that this diagram only illustrates unidirectional replication. In reality, the *VHA peer* can become the *VHA origin* when it receives a request for an object it doesn't have in cache. So in fact, replication can be bidirectional, or even omnidirectional.

Efficient replication

The main thing to remember here is that *VHA origin servers* don't push the objects directly, but instead announce new cache insertions via a `VHA_BROADCAST` request to all *peers*. This avoids broadcasting large objects across the network, potentially resulting in network saturation.

Another thing to remember is that when *peers* attempt to fetch these new objects, they don't get them from the backend web servers, but from the *VHA origin* server that announced it. This avoids unnecessary backend requests that may jeopardize stability, especially on a large *VHA cluster*.

Also important to know: the *VHA peer* will only acknowledge the `VHA_BROADCAST` request and send a corresponding `VHA_FETCH` request if it doesn't have that object already in cache. This means that replication only takes place for *cache misses* on the *peers*.

Efficiency also comes from the fact that *VHA* is designed for millisecond range replication. *VHA* already starts replicating as soon as the headers of the object are received. Even as data is streaming from the backend server to the *VHA origin*, the *VHA peers* start streaming the same content in parallel.

When does replication take place?

Not every *cache miss* will result in a `VHA_BROADCAST` request to the *VHA peers*. The *VHA* logic will qualify which backend responses can be replicated, based on a set of rules.

Here are some rules that exclude objects from being replicated:

- When `bereq.uncacheable` or `beresp.uncacheable` for the response equal `true`
- When replication for an object was explicitly skipped in *VCL*
- When the *TTL* of an object is equal to or less than the `min_ttl` *VHA* setting. *Three seconds* by default
- When the size of the object exceeds the `max_bytes` *VHA* setting. *25 MB* by default
- When the amount of *in-flight transactions* per second exceeds the `max_requests_sec` *VHA* setting. *200* by default

These rules make sense: if an object is short-lived or not cacheable at all, we really don't want to spend time and resources replicating it to the other nodes in the cluster. It's just not worth it.

The rules also ensure that replication doesn't overload the network or the *peer servers*. It does this by limiting the size of replicated object and by rate limiting the number of in-flight replications.

The thresholds that are used to quantify some of these restrictions can be configured. We'll talk about that soon.

Security

It is important that the `VHA_BROADCAST` and `VHA_FETCH` requests are secured. If the requests are tampered with, this can have a serious impact on the integrity and consistency of the data but also the stability of the platform.

The `VHA_BROADCAST` and `VHA_FETCH` requests are secured with a time-based *HMAC signature*. This means that replication messages are protected with a cryptographic signature that cannot be tampered with. This ensures the integrity of the data.

Because the *HMAC signature* is time-based, it cannot be duplicated or replayed.

The *VHA configuration* requires a cluster-wide unique *token*. This token is used as the signing key for the *HMAC signature* and is defined by the `token` *VHA setting*.

The validity of the *token* can be configured via the `token_ttl` *VHA setting*, which defaults to *two minutes*. This means that the *HMAC* signature is valid for two minutes. After that, the request is no longer considered valid.

These transactions can also be done over an *HTTPS connection*, ensuring that the outside world cannot decrypt the messages.

7.6.1 Installing VHA

VHA is only available for *Varnish Enterprise* and is packaged as `varnish-plus-ha`. Because it depends on the *broadcaster*, here's how you would install this on *Debian or Ubuntu* systems:

```
sudo apt-get install varnish-plus-ha varnish-broadcaster
```

This is the equivalent for *RHEL, CentOS, and Fedora*:

```
sudo yum install varnish-plus-ha varnish-broadcaster
```

The install will put the necessary *VHA VCL files* in the `/usr/share/varnish-plus/vcl/vha6` folder. It will also install the custom `vmod_vha` *VMOD*.

nodes.conf

The next step is to define your inventory inside `nodes.conf`. Here's an example from *chapter 6* when we first introduced the *broadcaster*:

```
[eu]
eu-varnish1 = http://varnish1.eu.example.com
eu-varnish2 = http://varnish2.eu.example.com
eu-varnish3 = http://varnish3.eu.example.com

[us]
us-varnish1 = http://varnish1.us.example.com
us-varnish2 = http://varnish2.us.example.com
us-varnish3 = http://varnish3.us.example.com
```

Unless defined otherwise, replication will happen across these six nodes.

Make sure you restart the *broadcaster* after you have changed your inventory.

VCL

And finally, it's a matter of including the necessary *VCL* and initializing *VHA*:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    call vha6_token_init;
}
```

And that's all it takes. Enabling *VHA* from your *VCL* code is surprisingly simple.

We already referred to *VHA settings* earlier and that *VCL example* shows how it is done using the `vha6_opts.set()` method. At the minimum a `token` setting should be defined. All other settings are optional.

The `vha6/vha_auto.vcl` include is loaded from the `vcl_path` directories. By default this is `/etc/varnish` and `/usr/share/varnish-plus/vcl`. As mentioned before the *VHA* files are located in `/usr/share/varnish-plus/vcl/vha6`.

The *VHA* files and the *VMODs* that are included are important and will hook in nicely with your existing *VCL*. It is a non-intrusive solution.

7.6.1 Configuring VHA

Although `token` is the only required setting, there are plenty of other *VHA* settings that can be configured. We've grouped the settings per topic.

Let's have a look.

Broadcaster settings

The *broadcaster* is a key component that initiates the replication. The default *broadcaster endpoint* is `http://localhost:8088`.

This implies that the *broadcaster* is hosted locally. This is a common pattern that makes using *VHA* quite simple. A potential downside is the fact that you have to manage the `nodes.conf` inventory on all *broadcaster* nodes. If for example, your inventory increases from a *five-node cluster* to a *six-node cluster*, you'll need to update the `nodes.conf` inventory on all six *broadcaster* nodes.

If you want the broadcaster to be centralized, you can configure *VHA* to send `VHA_BROADCAST` requests to a central endpoint.

Here's a list of settings you can edit to change the endpoint:

- `broadcaster_scheme`: the URL scheme to use. Defaults to `http` and can be set to `https`
- `broadcaster_host`: the hostname or IP address of the *broadcaster*. Defaults to `localhost`
- `broadcaster_port`: the TCP port on which the *broadcaster* is available. Defaults to `8088`

Imagine that your *broadcaster* is available through `http://broadcaster.example.com`. This would result in the following *VCL code*:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("broadcaster_scheme", "http");
    vha6_opts.set("broadcaster_host", "broadcaster.example.com");
    vha6_opts.set("broadcaster_port", "80");
    call vha6_token_init;
}
```

I
f we look back at our `nodes.conf` inventory, we have nodes in the *EU* and the *US*. We might want to restrict replication within a geographical region; otherwise we might experience latency because of the distance between the nodes.

Again, imagine that this is our inventory:

```
[eu]
eu-varnish1 = http://varnish1.eu.example.com
eu-varnish2 = http://varnish2.eu.example.com
eu-varnish3 = http://varnish3.eu.example.com

[us]
us-varnish1 = http://varnish1.us.example.com
us-varnish2 = http://varnish2.us.example.com
us-varnish3 = http://varnish3.us.example.com
```

We can set the broadcaster_group setting to eu within the *EU* to limit replication to only the EU nodes. Here's the *VCL* to do that:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("broadcaster_group", "eu");
    call vha6_token_init;
}
```

Origin settings

The VHA_BROADCAST request from the *VHA origin* to the *VHA peer* contains a vha6-origin request header. This header contains the endpoint that the *peer* should connect to when it sends out its VHA_FETCH request.

The *origin endpoint* is automatically generated by *VHA*, but you can override it if required.

The autodetection of the *origin scheme* and *origin port* is based on the port that is used for the incoming connection to the *VHA origin*. The *origin host* is based on the *VHA origin's* server.ip value.

The origin_scheme, origin, and origin_port settings can be used to override these automatically generated values.

Here's an example where we will force the *peer* to connect back to the *VHA origin* over *HTTPS*, even if original request was done over *HTTP*:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("origin_scheme", "https");
    vha6_opts.set("origin_port", "443");
    call vha6_token_init;
}
```

You can even redefine the *host* of the *VHA origin*. This allows you to potentially have *peers* request the new object from another system:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("origin", "vha-origin.example.com")
    call vha6_token_init;
}
```

TLS

TLS is crucial these days, and all *VHA* components can be configured to use *HTTPS* endpoints.

The `broadcaster_scheme` *VHA setting* can be set to `https` to ensure the communication between the *VHA origin* and the *broadcaster* is done over *TLS*. If you do this, please make sure the `broadcaster_port` setting also matches the *broadcaster*'s `https-port` value.

Here's some *VCL* that shows you how a locally hosted *broadcaster* can be configured to use *TLS*:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("broadcaster_scheme", "https");
    vha6_opts.set("broadcaster_port", "8443");
    call vha6_token_init;
}
```

Please keep in mind that your *broadcaster* instance has to be configured with *TLS* support to make this work.

The node definition in the *broadcaster*'s `nodes.conf` file can also start with `https://`. This forces the *broadcaster* to send `VHA_BROADCAST` requests to the *peers* over *HTTPS* and ensures that the `VHA_FETCH` and the fetched responses are sent over *HTTPS*.

Even if your `nodes.conf` inventory has a `http` scheme, or no scheme at all, it is still possible to enable *TLS/SSL* in *VHA*. It's a matter of setting `origin_scheme` to `https` and assigning the right port to `origin_port`.

We already featured an example with both of these settings. Let's make it a bit more interesting by making sure *self-signed certificates* can be used as well.

Here's the code:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("origin_scheme", "https");
    vha6_opts.set("origin_port", "443");
    vha6_opts.set("origin_ssl_verify_peer", "false");
    vha6_opts.set("origin_ssl_verify_host", "false");

    call vha6_token_init;
}
```

By setting both `origin_ssl_verify_peer` and `origin_ssl_verify_host` to `false`, the authenticity of a *TLS/SSL certificate* is ignored. That allows using certificates that were not issued by a *certificate authority*. In this case, the certificates were self-signed.

Disabling the *TLS/SSL* verification process can also be done for the *broadcaster*. However, the *broadcaster* plays two roles and that has an impact on the configuration.

To the *VHA origin*, the *broadcaster* acts as a server. To use self-signed certificates, you need to set `broadcaster_ssl_verify_peer` and `broadcaster_ssl_verify_host` to `false`. This ensures that *VHA* doesn't complain when the certificate is not authentic.

But the *broadcaster* also acts as the client towards the *VHA peers*. When `https` schemes are used in the `nodes.conf`, those endpoints need to have valid certificates as well, regardless of the *VHA settings*.

To make sure *peer and host verification* within the *broadcaster* is also disabled, you have to set the *broadcaster*'s `tls-verify` runtime configuration parameter to `NONE`.

Limits

As discussed earlier, *VHA* has put some limitations in place that impact how and when replication takes place.

The `min_ttl` setting, for example, defines what the minimum *TTL* of an inserted object must be before it is considered for replication. The default value is `3s`.

The `max_bytes` setting defines the upper limit in terms of payload size. By default this value is `25000000`. This means that for *HTTP responses* with a `Content-Length` header that exceeds *25000000 bytes*, the object will not be replicated.

The `max_requests_sec` defines the maximum number of *in-flight transactions per second* that are tolerated before replication is halted. The default value is `200`.

The `fetch_timeout` setting will limit the amount of time the *VHA peer* can spend waiting for the first byte of the object to be returned from the *VHA origin*. By default this is unlimited.

So let's throw these settings together into a single *VCL example*:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
    vha6_opts.set("min_ttl", "10s");
    vha6_opts.set("max_bytes", "100000000");
    vha6_opts.set("max_requests_sec", "1000");
    vha6_opts.set("fetch_timeout", "60s");

    call vha6_token_init;
}
```

These settings will prevent objects from being replicated if their *TTL* is lower than *ten seconds*, if the payload of the *HTTP response* is larger than *100 MB*, or if there are more than *1000 requests per second in-flight*.

When replication is active, the `VHA_FETCH` call is allowed to wait *one minute* before the first byte comes in. Otherwise the replication call fails.

Skipping replication

In the previous subsection, we talked about limits and when replication is prevented. But these are global settings. However, the `skip` setting allows you to skip replication on a *per-request basis*.

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
  call vha6_token_init;
}

sub vcl_backend_fetch {
    if(bereq.url ~ "^/video") {
        vha6_request.set("skip", "true");
    }
}
```

The example above will skip replication if the *backend request URL* matches the `^/video` regular expression pattern. Skipping replication on a per-request basis can only be done inside `vcl_backend_fetch` and `vcl_backend_response`.

Forcing an update

It is possible in *VHA* to force a transaction to update an existing object in cache. So even if the *VHA peers* have the object in cache, and would otherwise ignore the replication request, a new cache insertion can be forced.

The example below features a news website where all URLs that start with `/breaking-news` are forcefully replicated:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");
  call vha6_token_init;
}

sub vcl_backend_fetch {
    if(bereq.url ~ "^/breaking-news") {
        vha6_request.set("force_update", "true");
    }
}
```

7.6.1 Monitoring

VHA heavily relies on `vmod_kvstore` for managing options and storing metrics. With `varnishstat` these metrics can be visualized.

Here's an example of some *VHA6 stats* using `varnishstat`:

```
$ varnishstat -f *vha6_stats* -1
KVSTORE.vha6_stats.boot.broadcast_candidates  8  0.02 Broadcast can-
didates
KVSTORE.vha6_stats.boot.broadcasts            7  0.02 Successful
broadcasts
KVSTORE.vha6_stats.boot.fetch_peer            2  0.01 Broadcasts
which hit this peer node (fetches)
KVSTORE.vha6_stats.boot.fetch_origin          7  0.02 Fetches which
hit this origin node
KVSTORE.vha6_stats.boot.fetch_origin_deliver  7  0.02 Fetches which
were delivered from origin to the peer
KVSTORE.vha6_stats.boot.fetch_peer_insert     2  0.01 Fetches which
were successfully inserted
KVSTORE.vha6_stats.boot.error_fetch           0  0.00 Fetches which
encountered a network error
KVSTORE.vha6_stats.boot.error_fetch_insert    0  0.00 Fetches which
encountered an origin error
KVSTORE.vha6_stats.boot.broadcast_skip        1  0.00 Broadcast can-
didate has a VCL override
```

- The *first* column contains the *name* of the counter.
- The *second* column is the *current value* of the counter.
- The *third* column represents the *average per-second change* for that counter.
- The *fourth* column describes the *meaning* of the counter.
- Here's what we know, based on the output above:
- There are eight objects on this node that can be replicated.
- Seven objects were in fact replicated to *peer servers*.
- One object was skipped.
- For two objects this server was the *peer* and fetched the objects from the corresponding *VHA origin*.
- These two fetches were successfully stored in cache.
- The seven objects for which this node was the *VHA origin* were successfully fetched by the *peer* and delivered to the *peer*.

- No errors occurred while this node was fetching data from the corresponding *VHA origin*.

The `-1` varnishstat flag sends the counters to the standard output, instead of presenting the statistics as a continuously updated list.

7.6.1 Logging

The `varnishlog` program will also contain detailed logging information on *VHA* transactions.

The following `varnishlog` command will display logs for all transactions of which the *request method* starts with *VHA*. This includes `VHA_BROADCAST` requests on *VHA origin servers* and `VHA_FETCH` requests on *peer servers*:

```
varnishlog -g request -q "ReqMethod ~ VHA"
```

This command should be run on all nodes in your cluster, as it is unclear at what point a node is a *VHA origin* or a *VHA peer*. The output is extremely verbose. Let's just look at the `VHA_BROADCAST` request information:

```
-   ReqMethod      VHA_BROADCAST
-   ReqURL         /fed4a6511f33937f6de966469f98ad6f6ca1f-
9f4a2a41a24ef5d1abdde09980d
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: example.com
-   ReqHeader      Vha6-Date: Fri, 20 Nov 2020 13:38:12 GMT
-   ReqHeader      Vha6-Origin: https://192.168.0.5:443
-   ReqHeader      Vha6-Origin-Id: varnish1
-   ReqHeader      Vha6-Token: c5391fd44cba8ede76f4bd9b02d6c135e-
217608d63e5c07f3979ac854918162e
-   ReqHeader      Vha6-Url: /contact
-   ReqHeader      vha6-method: VHA_BROADCAST
-   ReqHeader      vha6-peer-id: varnish2
```

As you can see, the URL of the replicated object is `https://example.com/contact`. The `varnish1` server acted as the origin and can be reached through `https://192.168.0.5:443`. This information was received by the `varnish2` server, which acted as the *peer* for this transaction.

Similar information can be extracted for the `VHA_FETCH` call:

```
-   ReqMethod      VHA_FETCH
-   ReqURL         /fed4a6511f33937f6de966469f98ad6f6ca1f-
9f4a2a41a24ef5d1abdde09980d
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: example.com
-   ReqHeader      Vha6-Date: Fri, 20 Nov 2020 13:38:12 GMT
-   ReqHeader      Vha6-Origin: https://192.168.0.5:443
-   ReqHeader      Vha6-Origin-Id: varnish1
-   ReqHeader      Vha6-Token: c5391fd44cba8ede76f4bd9b02d6c135e-
217608d63e5c07f3979ac854918162e
-   ReqHeader      Vha6-Url: /contact
-   ReqHeader      vha6-method: VHA_FETCH
-   ReqHeader      vha6-peer-id: varnish2
```

This request was processed by `varnish1`, which is the *VHA origin* for this transaction. The information is very similar to the `VHA_BROADCAST` request and is an acknowledgement by *VHA peer*.

When the *VHA origin* responds to the *VHA peer* for the `VHA_FETCH` request, the following response information can be found in the *VSL* logs of the *VHA origin*:

```
-   RespProtocol   HTTP/1.1
-   RespStatus     200
-   RespReason     OK
-   RespHeader     Date: Fri, 20 Nov 2020 13:38:12 GMT
-   RespHeader     Content-Type: text/html
-   RespHeader     Content-Length: 612
-   RespHeader     Last-Modified: Tue, 27 Oct 2020 15:09:20 GMT
-   RespHeader     ETag: "5f983820-264"
-   RespHeader     X-Varnish: 23 65558
-   RespHeader     Age: 0
-   RespHeader     Via: 1.1 varnish (Varnish/6.0)
-   RespHeader     vha6-stevedore-ttl: 120.000s
-   RespHeader     vha6-stevedore-ttl-rt: 119.973s
-   RespHeader     vha6-stevedore-grace: 10.000s
-   RespHeader     vha6-stevedore-keep: 0.000s
-   RespHeader     vha6-stevedore-uncacheable: false
-   RespHeader     vha6-stevedore-storage: storage.s0
-   RespHeader     vha6-stevedore-age: 0
-   RespHeader     vha6-stevedore-insert: Fri, 20 Nov 2020 13:38:12
GMT
-   RespHeader     vha6-origin: varnish1
-   RespHeader     vha6-seal: b49ad1fcc95f1a740cddf65c8c9d653bd0c-
3737baa8bc684442d8988dc64fea6
```

This looks like a regular *HTTP response*, but it also includes some metadata in the form of `vha6-stevedore-*` headers.

7.6.1 Not using the broadcaster

It is technically possible to set up *VHA* without the *broadcaster*.

Although the *broadcaster* is essential for a multi-node cluster, it is not a hard requirement for a two-node cluster.

You can also send a `VHA_BROADCAST` request directly to your second node, as illustrated below:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");

    vha6_opts.set("broadcaster_host", "varnish2.example.com");
    vha6_opts.set("broadcaster_port", "443");
    vha6_opts.set("broadcaster_scheme", "https");

    call vha6_token_init;
}
```

We actually set the `broadcaster_host` to the host of your other *Varnish* node instead of relying on the *broadcaster* for this.

The same thing happens on your second node: setting the `broadcaster_host` to your first *Varnish* node. By doing this, you have bidirectional replication.

Here's the configuration for your second node:

```
vcl 4.1;

include "vha6/vha_auto.vcl";

sub vcl_init {
    vha6_opts.set("token", "secret123");

    vha6_opts.set("broadcaster_host", "varnish1.example.com");
    vha6_opts.set("broadcaster_port", "443");
    vha6_opts.set("broadcaster_scheme", "https");

    call vha6_token_init;
}
```

Although this solution is viable and can be considered a *high-availability* solution, it does take away a lot of flexibility. Unless you're certain that your two-node setup will remain a two-node setup, using the *broadcaster* is the recommended way to go.

7.6.1 Discovery

There is also an important operational question that hasn't been answered or raised:

How do you add or remove nodes from the cluster?

Failing nodes are supposed to be removed from the *broadcaster*'s inventory. This also applies to nodes that are removed when scaling in. And when a spike in demand is expected, extra nodes should be registered in the `node.conf` file.

Whenever an inventory change takes place, the `nodes.conf` file needs to be reprovisioned, and the `broadcaster` needs to be reloaded. By sending a `SIGHUP` signal to the *broadcaster*, the `nodes.conf` file is read, and the configuration reloaded.

Here's how the *broadcaster* process reports this:

```
DEBUG: Sighup notification received, reloading configuration
DEBUG: Reading configuration from /etc/varnish/nodes.conf
DEBUG: Done reloading configuration
```

You can do this manually by triggering a script to reprovision the inventory and reloading the *broadcaster*. But unless you have a proper discovery tool, you will not know when to change your `nodes.conf` file and what servers are to be considered.

Especially on environments with a dynamic inventory, like the cloud, this can become a serious challenge.

The varnish-discovery program

Luckily, *Varnish Enterprise* comes with the `varnish-discovery` program that dynamically provisions your *broadcaster*'s `nodes.conf`, and reloads the *broadcaster* service.

Varnish Discovery supports a couple of data providers that are used to provision the `nodes.conf` file.

These are the providers that are currently supported:

- Amazon Web Services
- Microsoft Azure
- DNS
- Kubernetes

With the exception of *DNS*, API calls are made to the data provider to retrieve inventory information.

For cloud platforms like *AWS* or *Azure*, the nodes that are associated with a specific autoscaling group are retrieved. For Kubernetes these are *pods* that are associated with an *endpoint list*.

The node information that was collected from the data provider is then flushed to the `nodes.conf` file and a `SIGHUP` signal is sent to the `broadcaster` process.

The `varnish-discovery` program will emit the following output when a change is detected:

```
Generating new nodefile /etc/varnish/nodes.conf (2020-11-23
16:14:17.4384255 +0000 UTC m=+58.052877601)
```

Installing varnish-discovery

Varnish Discovery is only available for *Varnish Enterprise* and is packaged as `varnish-plus-discovery`. Because this service also depends on the *broadcaster*, here's how you would install this on *Debian or Ubuntu* systems:

```
sudo apt-get install varnish-plus-discovery varnish-broadcaster
```

This is the equivalent for *RHEL, CentOS, and Fedora*:

```
sudo yum install varnish-plus-discovery varnish-broadcaster
```

A `systemd` service file is available in `/lib/systemd/system/varnish-discovery.service`. This is the default service definition of that file:

```
[Unit]
Description=Varnish Discovery
#After=network-online.target
#Requisite=network-online.target

[Service]
ExecStart=/usr/bin/varnish-discovery dns \
    --group example.com \
    --nodefile /etc/varnish/nodes.conf \
    --warnpid /run/vha-agent/vha-agent.pid

[Install]
WantedBy=multi-user.target
```

The standard configuration will most likely not work for you. But once the right runtime parameters are set, you can run the following commands to enable and start the service:

```
sudo systemctl enable varnish-discovery
sudo systemctl start varnish-discovery
```

Configuring varnish-discovery

Configuring `varnish-discovery` is done by setting the right runtime parameters. Let's take the default settings from the `systemd` service file and explain what they mean:

The first argument is the name of the provider to use. By default this is `dns`, but you can set this to `aws`, `azure`, or `kubernetes`.

The `--group` parameter is used to query the group that contains the nodes we want in our inventory. The meaning of *group* varies on the provider that is used:

- For `dns`, the `--group` parameter refers to the *DNS record* that contains the IP addresses of the *Varnish* nodes.
- For `aws`, the `--group` parameter refers to the *autoscaling group* that contains the *EC2 instances* on which *Varnish* is hosted.
- For `azure`, the `--group` parameter refers to the *virtual machine scale set* that contains the *virtual machines* on which *Varnish* is hosted.
- For `k8s`, the `--group` parameter refers to the *endpoint* that contains the *Varnish pods*.

The `--nodefile` parameter refers to the location of the `nodes.conf` file. This is where `varnish-discovery` sends its output.

The `--warnpid` parameter is the *PID file* that needs to be signaled when changes to the `nodes.conf` file have occurred. By default this is `/run/vha-agent/vha-agent.pid`, but this should be the *PID file* of the `broadcaster` service.

Please keep in mind that you probably have to add the `--pid` runtime parameter to the `systemd` configuration of your *broadcaster*. This ensures that the *PID file* is written out to the desired location.

There are some other parameters you can configure as well. Here's a quick overview:

- `--proto` is the backup protocol that will be used in `nodes.conf` in case a protocol wasn't returned from the data provider. Defaults to `http`
- `--port` is the backup port that is used when the port wasn't returned by the data provider. If omitted, the port is inferred from the protocol.
- `--ipv4` and `--ipv6` allow you to filter nodes on their IP protocol version. If omitted, both *IPv4* and *IPv6* addresses are allowed for nodes.
- `--once` will not continuously query the data provider, but instead will only query it once.
- `--every` is used to define the frequency with which the data provider is queried. Defaults to two seconds

Here's an example where we use some more runtime parameters:

```
/usr/bin/varnish-discovery dns \
    --group example.com \
    --nodefile /etc/varnish/nodes.conf \
    --warnpid /var/run/broadcaster.pid \
    --proto https \
    --port 444 \
    --ipv4 --every 10
```

This example will resolve the `example.com` hostname every *ten seconds* and will only retrieve *IPv4 addresses*. The nodes that are written to `/etc/varnish/nodes.conf` will be prefixed with `https://` and suffixed with `:444` for the port.

Once the `nodes.conf` file is written to disk, the *PID* inside `/var/run/broadcaster.pid` is used to send a `SIGHUP` signal to the *broadcaster*.

DNS

The `dns` provider will resolve a hostname that was provided by the `--group` parameter. The IP addresses resolved from the DNS call will end up in the `nodes.conf` file.

If an IP address matches the local machine, the local hostname is used instead of the IP address. You can disable this behavior by adding the `--no-hostname` runtime parameter.

This is a pretty basic solution that works on any platform. However, it is important to make sure the *TTL* of your DNS records is not too high. Otherwise local DNS resolvers might cache the value for longer than expected.

AWS

The *Amazon Web Services (AWS)* provider interacts with the *AWS API*. The `--group` parameter refers to the *autoscaling group* that is expected to be defined in your *AWS environment*.

The API call will retrieve the *private DNS name* and the *private IP address* for every node that is part of the *autoscaling group*. This information is then written to the `nodes.conf`.

The `-region` parameter sets the AWS region where the *autoscaling group* is defined. This is only necessary when the default region is not configured on your system via the `aws configure` command.

The `aws` program is a *CLI interface* for the *AWS API*. The `aws configure` command will assist with configuring the necessary environment variables for authenticating with the *AWS API*.

You can also define the following environment variables for authentication:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_DEFAULT_REGION`

Or you can define an `AWS_SHARED_CREDENTIALS_FILE` environment variable that refers to a shared credentials file.

Once the proper environment variables are set up, *VHA* will dynamically replicate its objects to *Varnish* servers that are part of this *autoscaling group*.

Azure

Azure is *Microsoft*'s cloud platform, and the `azure` provider within `varnish-discovery` is very similar to the `aws` one. The `--group` parameter refers to a *virtual machine scale set (VMSS)*, which is *Azure*'s version of an autoscaling group.

There are two custom parameters for *Azure* you can configure:

- `-resourcegroup`
- `-subscriptionid`

The *resource group* and *subscription* of the *VMSS* can be configured. This is just the fallback value in case these weren't configured via `az configure`.

The `az` program is also a *CLI interface*. It interacts with the *Azure API* and sets some environment variables that are used for authentication.

Here's the list of environment variables that are used for authentication:

- `AZURE_SUBSCRIPTION_ID`
- `AZURE_TENANT_ID`
- `AZURE_CLIENT_ID`
- `AZURE_CLIENT_SECRET`
- `AZURE_LOCATION_DEFAULT`
- `AZURE_BASE_GROUP_NAME`

By using `az configure`, these environment variables will be set for you, and the *VMSS* information is picked up.

If you're using *Azure*, this is an excellent way to dynamically scale your *Varnish* setup and make sure *VHA* keeps working.

Kubernetes

Kubernetes is a framework for automating the deployments, scaling, and management of containerized applications. It mostly uses *Docker containers*.

When *Varnish Enterprise* is run inside containers on a *Kubernetes cluster*, the `k8s` is there to figure out the various endpoints of a *service*. The `--group` parameter refers to a *service* inside a *Kubernetes cluster*. The service is the entrypoint to a set of *pods*, which contains the actual containers.

The *service* maps the *endpoints* of each *pod*, and our `k8s` provider will fetch those endpoints and provision them in `nodes.conf`.

Authentication with the *API* of that cluster is done based on a set of parameters:

- `--server`: the URL of the *API*. Defaults to `https://kubernetes`
- `--token`: the path to the token file. Defaults to `/var/run/secrets/kubernetes.io/serviceaccount/token`
- `--cacert`: the path to the CA certificate file. Defaults to `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`
- `--namespace`: the path to the namespace file. Defaults to `/var/run/secrets/kubernetes.io/serviceaccount/namespace`

These parameters are entirely optional, and the default values work perfectly fine if `varnish-discovery` runs inside a *Kubernetes pod*. The `/var/run/secrets/kubernetes.io/serviceaccount/` path is accessible within the *pod* and the corresponding files are located in there.

However, if `varnish-discovery` runs elsewhere, the *Kubernetes API* needs to be called externally, which might require the authentication parameters to be tuned.

Here's an example where custom parameters are used for *Kubernetes* authentication:

```
/usr/bin/varnish-discovery k8s \
    --group my-varnish-service \
    --nodefile /etc/varnish/nodes.conf \
    --warnpid /var/run/broadcaster.pid \
    --server https://kubernetes-api.example.com \
    --token /path/to/token \
    --cacert /path/to/ca.crt \
    --namespace /path/to/namespace
```

Behind the scenes `https://kubernetes-api.example.com` is set as the base URL, and the content of `/path/to/namespace` is used to determine the URL. If `/path/to/namespace` contains `my-namespace`, the URL will become the following:

```
https://kubernetes-api.example.com/api/v1/namespaces/my-namespace/
endpoints
```

Authentication is required to access this API call. This is done via a *bearer authentication token*. The value of this token is the content of the file that was referred to via the `--token` parameter. In our case this is `/path/to/token`.

The TLS certificates that are used to encrypt the HTTPS connection are self-signed. In order to correctly validate the server certificate, a custom *CA certificate* is passed. This is the certificate that was used to sign the server certificates.

If you were to try this yourself using `curl`, this would be the corresponding call:

```
curl -H "Authorization: Bearer $(head -n1 /path/to/token)" \
    --cacert /path/to/ca.crt \
    https://kubernetes-api.example.com/api/v1/namespaces/$(head -n1 /
path/to/namespace)/endpoints
```

And that's exactly what the `k8s` provider does behind the scenes.

7.7 Monitoring

Uptime, throughput and performance are some of the key requirements of any service. *Varnish* happens to be really good at these things. But despite its capabilities, there is no guarantee that *Varnish* will perform great at all times.

Just like any program, process or service, *monitoring tools* are required to check the availability and efficiency of *Varnish*.

Tools like *Nagios* or *Zabbix* are very popular monitoring tools that check the availability of services and send alerts when *warnings or critical errors* occur.

For an *HTTP service* like *Varnish*, a typical *HTTP check* is often used to monitor the availability of *Varnish*.

There are also *Varnish* plugins for both *Nagios* and *Zabbix*, which dig a bit deeper. These plugins use *Varnish*'s built-in counters to monitor conditions that go way beyond general availability.

In these monitoring sections, we won't be talking about *Nagios* or *Zabbix*. We'll primarily focus on *Varnish counters*, and how you can retrieve and visualize them.

7.7.1 Varnishstat

`varnishstat` is a program that can display the internal *Varnish* counters. Here's an extract of the output you may get when you run the command:

```
Uptime mgt:      0+00:01:27
Hitrate n:       10        14        14
Uptime child:    0+00:01:28
avg(n):    0.1968    0.2143    0.2143

    NAME                                       CURRENT         CHANGE
AVERAGE         AVG_10       AVG_100       AVG_1000
MGT.uptime                                    0+00:01:27
MAIN.uptime                                   0+00:01:28
MAIN.sess_conn                                    16395           0.00
186.31         219.27        231.37        231.37
MAIN.client_req                                   16395           0.00
186.31         219.27        231.37        231.37
MAIN.cache_hit                                    16291           0.00
185.12         219.27        231.37        231.37
MAIN.cache_miss                                     104           0.00
1.18           0.00          0.00          0.00
```

```
MAIN.backend_conn                                       47           0.00
0.53          0.00          0.00          0.00
MAIN.backend_reuse                                      58           0.00
0.66          0.00          0.00          0.00
MAIN.backend_recycle                                   105           0.00
1.19          0.00          0.00          0.00
MAIN.fetch_length                                      105           0.00
1.19          0.00          0.00          0.00
MAIN.pools                                               2           0.00
.            2.00          2.00          2.00
MAIN.threads                                           127           0.00
.          127.00        127.00        127.00
MAIN.threads_created                                   127           0.00
1.44          0.00          0.00          0.00
MAIN.busy_sleep                                         70           0.00
0.80          0.00          0.00          0.00
MAIN.busy_wakeup                                        70           0.00
0.80          0.00          0.00          0.00
MAIN.sess_queued                                        20           0.00
0.23          0.00          0.00          0.00
MAIN.n_object                                          105           0.00
.          105.00        105.00        105.00
MAIN.n_objectcore                                      187           0.00
.          187.00        187.00        187.00
MAIN.n_objecthead                                      187           0.00
.          187.00        187.00        187.00
MAIN.n_backend                                           2           0.00
.            2.00          2.00          2.00
MAIN.s_sess                                          16395           0.00
186.31        219.27        231.37        231.37
MAIN.s_fetch                                           104           0.00
1.18          0.00          0.00          0.00
```

You see the counters being listed with their absolute value, the change rate, and some averages.

- *Current* represents the current value of a counter, since the start of the `varnishd` process.
- *Change* reflects the average value per second of a counter since the last update interval.
- *Average* is the average value of the counter since the start of the `varnishd` process.
- *AVG_10* is the average value of the counter over the last ten update intervals.
- *AVG_100* is the average value of the counter over the last 100 update intervals.
- *AVG_1000* is the average value of the counter over the last 1000 update intervals.

Varnishstat options

The varnishstat program has a set of options that allow you to control the output.

If you run varnishstat without any options, you end up in *curses mode*, which displays a continuously updated list of counters.

By adding -1, as illustrated below, the output is returned via *standard output*:

```
varnishstat -1
```

The output is a lot more verbose and includes all counters, including the ones that have a zero value. Here's a very small extract from a very long list:

```
MGT.uptime                 141         0.99 Management process uptime
MGT.child_start              1         0.01 Child process started
MGT.child_exit               0         0.00 Child process normal exit
MGT.child_stop               0         0.00 Child process unexpected
exit
MGT.child_died               0         0.00 Child process died (sig-
nal)
MGT.child_dump               0         0.00 Child process core dumped
MGT.child_panic              0         0.00 Child process panic
MAIN.summs               73384       516.79 stat summ operations
MAIN.uptime                142         1.00 Child process uptime
MAIN.sess_conn           36469       256.82 Sessions accepted
MAIN.sess_drop               0         0.00 Sessions dropped
MAIN.sess_fail               0         0.00 Session accept failures
```

If you want to get the list of available counters with a description, but without the actual values, you can run the following command:

```
varnishstat -l
```

Because the output is so verbose, the -f option can be used to filter the output based on a *glob pattern*.

Here's an example where we only want to see the gauge values of the various *shared memory allocation (SMA)* stevedores:

```
varnishstat -f "SMA.*.g_*" -1
```

Here's the output:

```
SMA.s0.g_alloc              40          .   Allocations outstanding
SMA.s0.g_bytes        41445760          .   Bytes outstanding
SMA.s0.g_space        63411840          .   Bytes available
SMA.Transient.g_alloc           23          .   Allocations outstand-
ing
SMA.Transient.g_bytes         7176          .   Bytes outstanding
SMA.Transient.g_space            0          .   Bytes available
```

Because the storage wasn't explicitly named in this case, `s0` is the automatic name that is assigned to the storage. If multiple storage engines are configured, the number that is used increases.

What can we learn from these counters?

- The total storage size (`g_bytes + g_space`) is *100 MB*.
- `SMA.s0.g_bytes` indicates that *40 MB* of storage is in use.
- `SMA.s0.g_space` says that about *60 MB* is still available.
- The *transient storage* is not limited in size because of the `SMA.Transient.g_space` zero byte value.
- At this point, *7176 bytes* worth of data is stored in *transient storage*. This corresponds to the `SMA.Transient.g_bytes` counter.

Here's an example where we combine multiple filters:

```
varnishstat -1 -f "MAIN.cache_*" -f "MAIN.s_*" \
    -f "MAIN.n_object" -f "^*bytes" -f "^*pipe_*"
```

This command will print all the `MAIN.cache_` counters, all the `MAIN.s_`, and the `MAIN.n_object` counter. However, all counters that end with `bytes`, or contain `pipe_` will not be shown.

This results in the following output:

```
MAIN.cache_hit          26760        12.55 Cache hits
MAIN.cache_hit_grace        0         0.00 Cache grace hits
MAIN.cache_hitpass          0         0.00 Cache hits for pass
MAIN.cache_hitmiss       5112         2.40 Cache hits for miss
MAIN.cache_miss          5190         2.43 Cache misses
MAIN.n_object            2299          .   object structs made
MAIN.s_sess             37263        17.48 Total sessions seen
MAIN.s_pipe                 0         0.00 Total pipe sessions
seen
MAIN.s_pass              5284         2.48 Total passed requests
seen
MAIN.s_fetch            10474         4.91 Total backend fetches
initiated
MAIN.s_synth                0         0.00 Total synthetic re-
sponses made
```

And let's break it down:

- *26760 cache hits* have taken place so far. Currently the hit rate is at *12.55* hits per second.
- There are no *grace hits*, which means no objects were served passed their *TTL*.
- There are no *hit-for-passes* because the *VCL* defaults to *hit-for-miss* instead.
- *5112 hit-for-misses* took place at a rate of *2.40* per second.
- *5190 cache misses* occurred at a rate of *2.43* per second.
- A total number of *2299* objects are stored in cache at this point.
- *37263* HTTP sessions were established. This happens at a rate of *17.48* new sessions per second.
- There wasn't a single *pipe* request taking place.
- *5284* requests were *passed* to the backend. In this case because they were *POST requests*. This happened at a rate of *2.48* per second.
- In total *10474* backend fetches took place. On average this happens about *4.91 times per second*.
- No *synthetic responses* have occurred so far.

Other output formats

The output that `varnishstat -1` returns uses new lines to delimit counters and spaces to delimit columns for each counter. Pretty standard stuff.

But you can also return the output as *XML or JSON* data by using the `-x` or `-j` options.

The following command will return the *hit and miss counters* and return them in *XML format*:

```
varnishstat -f "MAIN.cache_hit" -f "MAIN.cache_miss" -x
```

This is the corresponding *XML output*:

```
<?xml version="1.0"?>
<varnishstat timestamp="2020-12-10T13:55:34">
    <stat>
        <name>MAIN.cache_hit</name>
        <value>45718</value>
        <flag>c</flag>
        <format>i</format>
        <description>Cache hits</description>
    </stat>
    <stat>
        <name>MAIN.cache_miss</name>
        <value>8949</value>
        <flag>c</flag>
        <format>i</format>
        <description>Cache misses</description>
    </stat>
</varnishstat>
```

The output adds a bit more context about the counters that are returned. The `<flag>` and `<format>` tags are especially helpful.

In this case both counters have a `<flag>c</flag>` tag, which implies that the counter is a regular counter, not a *gauge*. Its value can only increase, not decrease.

Both counters also have a `<format>i</format>` tag, which implies that the value is an integer.

But if we run `varnishstat -f "SMA.s0.g_bytes" -x` to get the current size of the *shared memory allocation stevedore*, this is the *XML* output you'd get:

```
<?xml version="1.0"?>
<varnishstat timestamp="2020-12-10T14:02:28">
    <stat>
        <name>SMA.s0.g_bytes</name>
        <value>18876</value>
        <flag>g</flag>
        <format>B</format>
        <description>Bytes outstanding</description>
    </stat>
</varnishstat>
```

The `<flag>g</flag>` specifies that this value is a *gauge*: it can increase and decrease. The format of this output returns a *byte value*, hence the `<format>B</format>` tag.

Let's run the same commands again, and use the `-j` option to return *JSON* output:

```
varnishstat -f "MAIN.cache_hit" -f "MAIN.cache_miss" -j
```

This is the corresponding *JSON* output:

```
{
  "timestamp": "2020-12-10T13:55:34",
  "MAIN.cache_hit": {
    "description": "Cache hits",
    "flag": "c",
    "format": "i",
    "value": 45718
  },
  "MAIN.cache_miss": {
    "description": "Cache misses",
    "flag": "c",
    "format": "i",
    "value": 8949
  }
}
```

Again, you see the `flag` and `format` fields, and if we run `varnishstat -f "SMA.s0.g_bytes" -j`, we'll see *gauges* and *byte counts*:

```
{
  "timestamp": "2020-12-10T14:02:28",
  "SMA.s0.g_bytes": {
    "description": "Bytes outstanding",
    "flag": "g",
    "format": "B",
    "value": 18876
  }
}
```

Curses mode

Curses mode is actually the standard mode when you run `varnishstat`. Only when you use the `-1`, `-x`, or `-j` mode, you'll end up not using curses.

The very first `varnishstat` example we showed you was in *curses mode*. The output can be quite verbose, but just like in the other output modes, you can use *glob filters* to reduce the noise.

Let's run the following command in *curses mode*:

```
varnishstat -f "VBE.*"
```

This one will list some stats for every registered backend. The more backends you defined, the more output you'll get.

Here's some output while using a single backend:

```
Uptime mgt:      0+00:05:24                                 Hitrate n:
10       51        51
Uptime child:    0+00:05:25                                     avg(n):
0.7171   0.5066   0.5066

    NAME                       CURRENT        CHANGE        AVERAGE
AVG_10
MGT.uptime                  0+00:05:24
MAIN.uptime                 0+00:05:25
MAIN.cache_hit                  122288        107.80          376.27
516.95
MAIN.cache_miss                   6931          1.00           21.33
26.08
VBE.boot.default.bere...          3.28M          1.77K          10.32K
11.73K
VBE.boot.default.bere...         62.68K         53.90          197.50
211.65
```

```
VBE.boot.default.bere...         2.73M          1.33K          8.59K
9.86K
VBE.boot.default.bere...         9.34M          4.79K         29.42K
33.60K
VBE.boot.default.conn               3          2.99              .
1.27
VBE.boot.default.req            14108          9.98          43.41
49.79
```

> We deliberately kept the window size quite small while grabbing this output. This results in the `AVG_100` and `AVG_1000` columns being hidden.

While we only filtered for counters matching the `VBE` category, some extra `MGT` and `MAIN` counters ended up in the output as well.

Curses mode has some key bindings that allow you to manipulate the view.

When you use the `<UP>` or `<DOWN>` keys, you can navigate through the counter list. At the bottom of your windows, you'll see some extra information when a specific counter is highlighted. Here's what you see when the `VBE.boot.default.bereq_hdrbytes` counter is selected:

```
    VBE.boot.default.bereq_hdrbytes
INFO  1-10/10
Request header bytes:
        Total backend request header bytes sent
```

You can also use `<PAGEUP>` or `<b>`, and `<PAGEDOWN>` or `<SPACE>` to skip through page by page. And as expected, the `<HOME>` and `<END>` keys are there to take you to the top and bottom of the list.

When you press the `<d>` key, you can toggle between showing and hiding counters with a zero value. As you can see, some extra counters appear in the output:

```
    NAME                                    CURRENT        CHANGE
AVERAGE        AVG_10       AVG_100      AVG_1000
MGT.uptime                              0+00:14:03
MAIN.uptime                             0+00:14:04
MAIN.cache_hit                              248289          0.00
294.18           0.00         53.83         258.67
MAIN.cache_miss                              12872          0.00
15.25           0.00          2.57          12.21
VBE.boot.default.happy                  0000000000
________________________
VBE.boot.default.bereq_hdrbytes               6.05M         0.00
7.35K          0.00          1.23K         5.84K
VBE.boot.default.bereq_bodybytes            115.40K         0.00
140.01           0.00         23.23         110.58
VBE.boot.default.beresp_hdrbytes              5.05M         0.00
6.12K          0.00          1.03K         4.87K
VBE.boot.default.beresp_bodybytes            17.27M         0.00
20.95K           0.00          3.51K         16.65K
VBE.boot.default.pipe_hdrbytes                  0             .
.             0.00          0.00          0.00
VBE.boot.default.pipe_out                       0             .
.             0.00          0.00          0.00
VBE.boot.default.pipe_in                        0             .
.             0.00          0.00          0.00
VBE.boot.default.conn                           0           0.00
.             0.00          0.05          0.23
VBE.boot.default.req                          26086         0.00
30.91           0.00          5.18          24.57
VBE.boot.default.unhealthy                      0             .
.             0.00          0.00          0.00
VBE.boot.default.busy                           0             .
.             0.00          0.00          0.00
VBE.boot.default.fail                           0             .
.             0.00          0.00          0.00
VBE.boot.default.fail_eacces                    0             .
.             0.00          0.00          0.00
VBE.boot.default.fail_eaddrnota...              0             .
.             0.00          0.00          0.00
VBE.boot.default.fail_econnrefused              0             .
.             0.00          0.00          0.00
VBE.boot.default.fail_enetunreach               0             .
.             0.00          0.00          0.00
VBE.boot.default.fail_etimedout                 0             .
.             0.00          0.00          0.00
VBE.boot.default.fail_other                     0             .
.             0.00          0.00          0.00
VBE.boot.default.helddown                       0             .
.             0.00          0.00          0.00
```

When you press the `<e>` key, you can toggle the scaling values between the *byte count* and the standard value that includes a suffix.

This is the standard scaling of values:

```
VBE.boot.default.beresp_bodybytes    18.22M    0.00    18.17K
```

Megabytes and kilobytes are used as a unit of size. When you press `<e>`, this is what you'll see for this counter:

```
VBE.boot.default.beresp_bodybytes    19105054.72    0.00    18606.08
```

There are also some *hidden counters* that are only there for debug purposes. By pressing the `<v>` key, you cycle through the verbosity levels.

- `info` is the standard verbosity level.
- `diag` is the next level, which also shows *diagnostic counters*.
- `debug` is the final level, which also shows *debug counters*.

You can test this by running the following command:

```
varnishstat -f "MAIN.n_obj_purged" -f "MAIN.esi_warnings" -f "MAIN.
hcb_nolock"
```

- `MAIN.n_obj_purged` is an informational counter that will be visible by default.
- By pressing `<v>`, you'll end up in *diagnostic mode* and the `MAIN.esi_warnings` counter will appear.
- By pressing `<v>` again, you'll end up in *debug mode* and the `MAIN.hcb_nolock` counter will also appear.

> Make sure you press `<d>` as well for this example because some of these counters may have a zero value, and would otherwise remain hidden.

By pressing the `<+>` and `<->` keys, you can increase and decrease the refresh interval. The standard refresh interval is *one second*. Increasing or decreasing the interval is done *per 0.1 second*.

And finally you can press `<q>` to exit `varnishstat`.

7.7.2 Varnish counters

The *Varnish* source code has hundreds of counters that keep track of measurable events. Here's some example code that increases the *cache hit counter* when a cached object is served:

```
wrk->stats->cache_hit++;
```

As mentioned earlier, `varnishstat` is used to visualize these counters.

We already showed you a number of counters, but in this section we'll highlight the most important counters, grouped per category.

Main counters

There are about 130 *main counters*. They are prefixed with `MAIN.` and give you a basic understanding of what is happening.

Here's a first group of counters:

```
MAIN.uptime                    Child process uptime
MAIN.sess_conn                 Sessions accepted
MAIN.client_req                Good client requests received
MAIN.cache_hit                 Cache hits
MAIN.cache_hit_grace           Cache grace hits
MAIN.cache_hitpass             Cache hits for pass
MAIN.cache_hitmiss             Cache hits for miss
MAIN.cache_miss                Cache misses
MAIN.s_pipe                    Total pipe sessions seen
MAIN.s_pass                    Total passed requests seen
MAIN.s_fetch                   Total backend fetches initiated
MAIN.s_synth                   Total synthetic responses made
```

The fact that a description is part of the output doesn't require much elaboration. The `MAIN.cache_*` and `MAIN.s_*` counters give insight into the *finite state machine*, and which paths are the most common.

You can have a pretty decent view of the *offload* using the counters. This means the number of requests that were offloaded from the *origin*. For the requests that weren't offloaded, you can see why that was the case.

These are all basic counters, not gauges: they increase and cannot decrease. For analysis purposes the absolute values don't always matter. It's the change rate that matters.

There are also backend counters, session counters, threading counters, client counters, fetch counters, ban counters, workspace counters. A lot of counters, really.

Before we go to the next category, we just want to show you a set of counters that start with `MAIN.n_*`. Here's an extract of that list:

```
MAIN.n_object                   object structs made
MAIN.n_backend                  Number of backends
MAIN.n_expired                  Number of expired objects
MAIN.n_lru_nuked                Number of LRU nuked objects
MAIN.n_purges                   Number of purge operations executed
MAIN.n_obj_purged               Number of purged objects
```

The `MAIN.n_object` is a very common one to use. It indicates how many objects are stored in cache. The other counters are also pretty straightforward.

Management process counters

The counters of the *management process* are prefixed with `MGT.`, and only the `MGT.uptime` counter is displayed in the standard mode.

Here's the complete list of *management process counters*:

```
MGT.uptime                      Management process uptime
MGT.child_start                 Child process started
MGT.child_exit                  Child process normal exit
MGT.child_stop                  Child process unexpected exit
MGT.child_died                  Child process died (signal)
MGT.child_dump                  Child process core dumped
MGT.child_panic                 Child process panic
```

When a counter like `MGT.child_panic` increases, you know something is wrong.

Malloc stevedore counters

When you run *Varnish* using the `-s malloc` runtime parameter, the `SMA` counters will keep track of memory allocation and memory consumption.

If you only use a single unnamed *malloc stevedore*, the standard name will be `s0`. If you use two of them, both `s0` and `s1` counters will exist.

However, if you use `-s mem=malloc` to name your stevedore that name will be available on the counters instead of `s0`.

Here's the list of `SMA` counters:

```
SMA.s0.c_req                  Allocator requests
SMA.s0.c_fail                 Allocator failures
SMA.s0.c_bytes                Bytes allocated
SMA.s0.c_freed                Bytes freed
SMA.s0.g_alloc                Allocations outstanding
SMA.s0.g_bytes                Bytes outstanding
SMA.s0.g_space                Bytes available
SMA.Transient.c_req           Allocator requests
SMA.Transient.c_fail          Allocator failures
SMA.Transient.c_bytes         Bytes allocated
SMA.Transient.c_freed         Bytes freed
SMA.Transient.g_alloc         Allocations outstanding
SMA.Transient.g_bytes         Bytes outstanding
SMA.Transient.g_space         Bytes available
```

The counters that start with `c_` are regular counters, and the ones that start with `g_` are gauges.

The most common ones you'll use are `SMA.s0.g_bytes` and `SMA.s0.g_space` to view memory usage and available space. These counters are also available for the *transient storage* in the form of `SMA.Transient.g_bytes` and `SMA.Transient.g_space`.

By default `SMA.Transient.g_space` will be zero because *transient storage* is usually unbounded.

Backend counters

We've referred to the `MAIN.backend_*` counters earlier. These counters paint a general picture of your backends. But there are also `VBE` counters that focus on individual backends. The more backends you have, the longer the output.

Here's a collection of counters for backends defined in the standard *VCL* configuration that was loaded at boot time. Hence the `boot` prefix. As there is only one backend named `default` in that *VCL* configuration, the `VBE.boot.default.*` prefix contains all the required counters:

```
VBE.boot.default.happy         Happy health probes
VBE.boot.default.bereq_hdrbytes Request header bytes
VBE.boot.default.bereq_bodybytes Request body bytes
VBE.boot.default.beresp_hdrbytes Response header bytes
VBE.boot.default.beresp_bodybytes Response body bytes
VBE.boot.default.pipe_hdrbytes Pipe request header bytes
VBE.boot.default.pipe_out      Piped bytes to backend
VBE.boot.default.pipe_in       Piped bytes from backend
VBE.boot.default.conn          Concurrent connections to backend
VBE.boot.default.req           Backend requests sent
VBE.boot.default.unhealthy     Fetches not attempted due to backend
being unhealthy
VBE.boot.default.busy          Fetches not attempted due to backend
being busy
VBE.boot.default.fail          Connections failed
VBE.boot.default.fail_eacces   Connections failed with EACCES or
EPERM
VBE.boot.default.fail_eaddrnotavail Connections failed with EADDRNOT-
AVAIL
VBE.boot.default.fail_econnrefused Connections failed with ECONNRE-
FUSED
VBE.boot.default.fail_enetunreach Connections failed with ENETUNREACH
VBE.boot.default.fail_etimedout Connections failed ETIMEDOUT
VBE.boot.default.fail_other    Connections failed for other reason
VBE.boot.default.helddown      Connection opens not attempted
```

Some of these counters are *byte counters*; some of them regular ones. The `VBE.boot.default.happy` is an odd one, though.

When we run `varnishstat -x -f VBE.boot.default.happy` to see the details about this counter, this is the output we get:

```
<?xml version="1.0"?>
<varnishstat timestamp="2020-12-11T10:56:58">
    <stat>
        <name>VBE.boot.default.happy</name>
        <value>18446744073709551615</value>
        <flag>b</flag>
        <format>b</format>
        <description>Happy health probes</description>
    </stat>
</varnishstat>
```

This counter is not a regular one, nor is it a gauge. It's a *bitmap*. The `1` represents a successful health check. The `0` of the bitmap is a failed health check.

The absolute integer value doesn't really matter; the change rate does. When we see this counter in *curses mode*, it makes a lot more sense:

```
VBE.boot.default.happy ffffffffff VVVVVVVVVVVVVVVVVV______
```

We can conclude that the last five checks failed. This is visualized by `______`.

The other counters can be used to get more information about the *offload* of a specific backend. And if health checks start failing, the `VBE.boot.default.fail_*` counters can tell you why.

MSE counters

Varnish Enterprise adds three extra categories of counters for *MSE* alone. When you run *MSE* using a single *book* and a single *store*, you already have more than 100 counters available. Most of which are *diagnostic and debug* counters.

Here's some the output when running `varnishstat -f MSE*`:

```
    NAME                                        CURRENT        CHANGE
AVERAGE        AVG_10        AVG_100       AVG_1000
MGT.uptime                                   0+00:09:24
MAIN.uptime                                  0+00:09:25
MAIN.cache_hit                                    13002         16.97
23.01         18.04          3.13          1.62
MAIN.cache_miss                                     655          1.00
1.16          1.46          0.33          0.17
MSE.mse.c_req                                         0             .
.            0.00          0.00          0.00
MSE.mse.c_fail                                        0             .
.            0.00          0.00          0.00
MSE.mse.c_bytes                                  80.58M         1.05K
146.04K      1020.08K       353.78K       194.94K
MSE.mse.c_freed                                  41.05M         1.05K
74.40K          2.22K       412.70        213.75
MSE.mse.g_alloc                                      41          0.00
.           38.82         38.53         38.53
MSE.mse.g_bytes                                  39.53M         -1.00
.           37.43M        37.16M        37.16M
MSE.mse.g_space                                 212.43M          0.00
.          221.63M       246.76M       249.25M
MSE.mse.n_lru_nuked                                   0             .
.            0.00          0.00          0.00
MSE.mse.n_vary                                        0             .
.            0.00          0.00          0.00
MSE.mse.c_memcache_hit                            4.29G        33.54M
7.77M        36.68M         6.54M         3.39M
```

```
MSE.mse.c_memcache_miss                           0            .
.            0.00          0.00          0.00
MSE_BOOK.book.n_vary                              0            .
.            0.00          0.00          0.00
MSE_BOOK.book.g_bytes                        36.00K         0.00
.           33.53K        25.68K        24.87K
MSE_BOOK.book.g_space                        99.96M         0.00
.           99.97M        99.97M        99.98M
MSE_BOOK.book.g_banlist_bytes                     0            .
.            0.00          0.00          0.00
MSE_BOOK.book.g_banlist_space              1020.00K         0.00
.         1020.00K      1020.00K      1020.00K
MSE_BOOK.book.g_banlist_database                 16         0.00
.           16.00         16.00         16.00
MSE_STORE.store.g_aio_running                     0            .
.            0.00          0.00          0.00
MSE_STORE.store.g_aio_running_bytes               0            .
.            0.00          0.00          0.00
MSE_STORE.store.c_aio_finished                  101         0.00
0.18            1.03          0.35          0.19
MSE_STORE.store.c_aio_finished_bytes         79.14M         0.00
143.44K          1.02M       353.92K       194.44K
MSE_STORE.store.g_aio_queue                       0            .
.            0.00          0.00          0.00
MSE_STORE.store.c_aio_queue                       0            .
.            0.00          0.00          0.00
MSE_STORE.store.c_aio_write_queue_o...            0            .
.            0.00          0.00          0.00
MSE_STORE.store.g_objects                        20         0.00
.           18.94         18.80         18.80
MSE_STORE.store.g_alloc_extents                  20         0.00
.           18.94         18.80         18.80
MSE_STORE.store.g_alloc_bytes                39.53M         0.00
.           37.43M        37.16M        37.16M
MSE_STORE.store.g_free_extents                    1         0.00
.            1.00          1.00          1.00
MSE_STORE.store.g_free_bytes                984.46M         0.00
.          993.45M      1018.77M      1021.32M
```

There are three distinct counter categories related to *MSE*:

- `MSE`: the memory caching counters of *MSE*
- `MSE_BOOK`: the persistence metadata counters per *book*
- `MSE_STORE`: the persistence counters per *store*

The `MSE.mse.g_bytes` and `MSE.mse.g_space` counters are very straightforward: they let you know how much object storage memory is in use, and how much is left for storing new objects.

The `MSE.mse.n_lru_nuked` lets us know how many objects were removed from cache because of lack of space.

The `MSE.mse.c_memcache_hit` and `MSE.mse.c_memcache_miss` represent the amount of *hits* and *misses* on the *MSE memory cache*.

Our test configuration only has one book named `book`, and one store named `store`. Hence the `MSE_BOOK.book.*` and `MSE_STORE.store.*` counters.

The `MSE_BOOK.book.g_bytes` and `MSE_BOOK.book.g_space` come as no surprise, and require no additional information.

In terms of metadata, the `MSE_BOOK.book.n_vary` holds the number of *cache variations* for that book. The `MSE_BOOK.book.g_banlist_bytes` and `MSE_BOOK.book.g_banlist_space` refer to *ban list journal* usage, whereas `MSE_BOOK.book.g_banlist_database` refers to persisted bans.

The `MSE_STORE.store.g_objects` counter counts the number of objects stored in a store named `store`. That sounds quite poetic, doesn't it?

And in the end, `MSE_STORE.store.g_alloc_bytes` counts the number of *bytes in allocated extents*, and `MSE_STORE.store.g_free_bytes` counts the *number of free bytes in unallocated extents*.

What you've seen so far are only the basic counters. If you increase the verbosity, there are more interesting counters available.

The *waterlevel* counters are particularly interesting, as they paint a picture of the *continuous free space* that is guaranteed by *MSE*:

```
MSE_BOOK.book.g_waterlevel_queue Number of threads queued waiting for
database space
MSE_BOOK.book.c_waterlevel_queue Number of times a thread has been
queued waiting for database space
MSE_BOOK.book.c_waterlevel_runs Number of times the waterlevel purge
thread was activated
MSE_BOOK.book.c_waterlevel_purge Number of objects purged to achieve
database waterlevel
MSE_STORE.store.g_waterlevel_queue Number of threads queued waiting
for store space
MSE_STORE.store.c_waterlevel_queue Number of times a thread has been
queued waiting for store space
MSE_STORE.store.c_waterlevel_purge Number of objects purged to
achieve store waterlevel
```

And then there's the *anti-fragmentation counters*. Here's a small extract of these counters:

```
MSE_STORE.store.g_alloc_small_extents Number of allocation extents
smaller than 16k
MSE_STORE.store.g_alloc_small_bytes Number of bytes in allocation ex-
tents smaller than 16k
MSE_STORE.store.g_alloc_16k_extents Number of allocation extents be-
tween 16k and 32k
MSE_STORE.store.g_alloc_16k_bytes Number of bytes in allocation ex-
tents between 16k and 32k
MSE_STORE.store.g_alloc_32k_extents Number of allocation extents be-
tween 32k and 64k
MSE_STORE.store.g_alloc_32k_bytes Number of bytes in allocation ex-
tents between 32k and 64k
MSE_STORE.store.g_free_small_extents Number of free extents smaller
than 16k
MSE_STORE.store.g_free_small_bytes Number of bytes in free extents
smaller than 16k
MSE_STORE.store.g_free_16k_extents Number of free extents between 16k
and 32k
MSE_STORE.store.g_free_16k_bytes Number of bytes in free extents be-
tween 16k and 32k
MSE_STORE.store.g_free_32k_extents Number of free extents between 32k
and 64k
MSE_STORE.store.g_free_32k_bytes Number of bytes in free extents be-
tween 32k and 64k
```

The full list goes way up and handles 11 different kinds of extents, each with their own counters. It's impossible to cover them all, but at least you know what is out there.

KVStore counters

When using `vmod_kvstore`, you can actually send custom counters to the *Varnish Shared Memory*, which can be used by `varnishstat`.

Imagine the following *VCL* code:

```
vcl 4.1;

import kvstore;
import std;

sub vcl_init{
    new stats = kvstore.init();
}
```

```
sub vcl_recv {
    stats.counter("request_method_" + std.tolower(req.method), 1,
varnishstat = true, "The number of HTTP " + req.method + " re-
quests");
}
```

This code tracks the usage of the different *request methods* and stores them with the `request_method_` prefix. By setting the `varnishstat` argument to `true` in the `.counter()` function, these counters can be displayed via `varnishstat`.

If we were to run `varnishstat -1 -f KVSTORE.*`, we could end up seeing the following output:

```
KVSTORE.stats.boot.__keys                3   0.02 Number of keys
KVSTORE.stats.boot.request_method_get   816   4.23 The number of HTTP
GET requests
KVSTORE.stats.boot.request_method_post   53   0.27 The number of HTTP
POST requests
KVSTORE.stats.boot.request_method_head    7   0.04 The number of HTTP
HEAD requests
```

Because we named our *KVStore* object `stats`, and our *VCL configuration* is labeled as `boot`, the `KVSTORE.stats.boot` prefix is going to appear in `varnishstat`.

Please note that `KVSTORE.*` counters aren't visible by default when running in *curses mode*. You either have to increase the verbosity or just use `-1`, `-x`, or `-j`.

Another interesting detail is the fact that there's a *comment* argument where you can add more context and meaning to the counter you are returning.

7.7.3 Prometheus

`varnishstat` is a great tool. But if you have 50 *Varnish* servers to monitor, you're going to have a lot of work. One way or the other, all these counters need to be centralized into a single database, and you're also going to want to visualize them using a dashboard.

There are many ways to do this, but we tend to prefer *Prometheus* for this.

Prometheus is an open-source monitoring and alerting tool that ingests timeseries data and has a built-in querying language called *PromQL* to retrieve metrics.

Data ingestion is based on an *HTTP pull model*. As long as the service you're monitoring has an *HTTP endpoint* that exposes metrics in the right format, *Prometheus* can pull that data and store it in its database. We call these services *exporters*. Besides the standard *node exporter* to retrieve global server metrics, there are many *custom exporters* to expose custom metrics.

There are a lot of integrations with third-party tools for visualization and data retrieval. *Grafana* is one of those integrations. It allows you to create dashboards based on the *PromQL* syntax.

There's only one piece of the puzzle that is missing: an *exporter for Varnish*.

Varnish Exporter

There are a couple of *Varnish exporters* in the wild, and some of them are really good, such as https://github.com/jonnenauha/prometheus_varnish_exporter.

It is a project written in *Go* and is easy to build. It leverages the `varnishstat` binary and exposes the output as *Prometheus metrics*.

Here are the options for the exporter:

```
$ prometheus_varnish_exporter --help

Usage of prometheus_varnish_exporter:
  -N string
        varnishstat -N value.
  -docker-container-name string
        Docker container name to exec varnishstat in.
  -exit-on-errors
        Exit process on scrape errors.
  -n string
        varnishstat -n value.
  -no-exit
        Deprecated: see -exit-on-errors
  -raw
        Raw stdout logging without timestamps.
  -test
        Test varnishstat availability, prints available metrics and
exits.
  -varnishstat-path string
        Path to varnishstat. (default "varnishstat")
  -verbose
        Verbose logging.
  -version
        Print version and exit
  -web.health-path string
```

```
        Path under which to expose healthcheck. Disabled unless con-
figured.
  -web.listen-address string
        Address on which to expose metrics and web interface. (de-
fault ":9131")
  -web.telemetry-path string
        Path under which to expose metrics. (default "/metrics")
  -with-go-metrics
        Export go runtime and http handler metrics
```

Debian and *Ubuntu* provide packages for `prometheus_varnish_exporter`. It makes installation easier and comes out of the box with a *Systemd service file*.

Prometheus will call the exporter over *HTTP* via the `http://<hostname>:9131/metrics` endpoint to retrieve the metrics.

If your *Varnish* server runs inside a *Docker container* and your exporter doesn't, you can even use the `prometheus_varnish_exporter -docker-container-name <name>` command to capture the `varnishstat` output from a *container*.

The output this *Varnish exporter* generates is extremely verbose, so here's an extract of some of the metrics:

```
# HELP varnish_main_cache_hit Cache hits
# TYPE varnish_main_cache_hit counter
varnish_main_cache_hit 3055
# HELP varnish_main_cache_hit_grace Cache grace hits
# TYPE varnish_main_cache_hit_grace counter
varnish_main_cache_hit_grace 20
# HELP varnish_main_cache_hitmiss Cache hits for miss
# TYPE varnish_main_cache_hitmiss counter
varnish_main_cache_hitmiss 166
# HELP varnish_main_cache_hitpass Cache hits for pass
# TYPE varnish_main_cache_hitpass counter
varnish_main_cache_hitpass 0
# HELP varnish_main_cache_miss Cache misses
# TYPE varnish_main_cache_miss counter
varnish_main_cache_miss 187
# HELP varnish_sma_g_bytes Bytes outstanding
# TYPE varnish_sma_g_bytes gauge
varnish_sma_g_bytes{type="s0"} 4.144576e+07
varnish_sma_g_bytes{type="transient"} 2184
# HELP varnish_sma_g_space Bytes available
# TYPE varnish_sma_g_space gauge
varnish_sma_g_space{type="s0"} 6.341184e+07
varnish_sma_g_space{type="transient"} 0
```

Prometheus will call the *Varnish Exporter* endpoint, read the data, and store the *300+* metrics in its database.

Telegraf

Telegraf is an open-source server agent that collects metrics. It has a wide range of input plugins and even has a native *Varnish* plugin. It's similar to `prometheus_varnish_exporter` but is more flexible.

Telegraf metrics aren't restricted to *Prometheus* and can be used by various monitoring systems. However, it does make sense in our context to expose *Telegraf metrics* in a *Prometheus format*.

Here's an example of a *Telegraf* configuration file:

```
[global_tags]
[agent]
  interval = "10s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000
  collection_jitter = "0s"
  flush_interval = "10s"
  flush_jitter = "0s"
  precision = ""
  hostname = ""
  omit_hostname = false
 [[outputs.prometheus_client]]
  listen = ":9273"
  path = "/metrics"
 [[inputs.varnish]]
   binary = "/usr/bin/varnishstat"
   stats = ["MAIN.cache_hit", "MAIN.cache_miss", "MAIN.n_object"]
```

The `[[outputs.prometheus_client]]` directive will set the necessary parameters for *Prometheus* to pull the metrics. The `[[inputs.varnish]]` directive will set parameters for metrics retrieval based on `varnishstat`.

A big difference in comparison to the *Varnish Exporter* is that you have to define which metrics you want to retrieve. In this example, we're collecting `MAIN.cache_hit`, `MAIN.cache_miss`, and `MAIN.n_object`. But it is also possible to specify a *glob pattern* to retrieve multiple metrics at once.

Running *Telegraf* can be done via the `telegraf --config /path/to/telegraf.conf`. But you might be better off running this as a *Systemd* service. If you install *Telegraf* through the packages provided by *InfluxData*, you'll already have a *Systemd unit file*.

These are the *Varnish* metrics coming out of *Telegraf* through `http://<hostname>:9273/metrics`:

```
# HELP varnish_cache_hit Telegraf collected metric
# TYPE varnish_cache_hit untyped
varnish_cache_hit{host="varnish",section="MAIN"} 0
# HELP varnish_cache_miss Telegraf collected metric
# TYPE varnish_cache_miss untyped
varnish_cache_miss{host="varnish",section="MAIN"} 0
# HELP varnish_n_object Telegraf collected metric
# TYPE varnish_n_object untyped
varnish_n_object{host="varnish",section="MAIN"} 0
```

But as mentioned, you can specify which metrics you want to collect in the *Telegraf* configuration file.

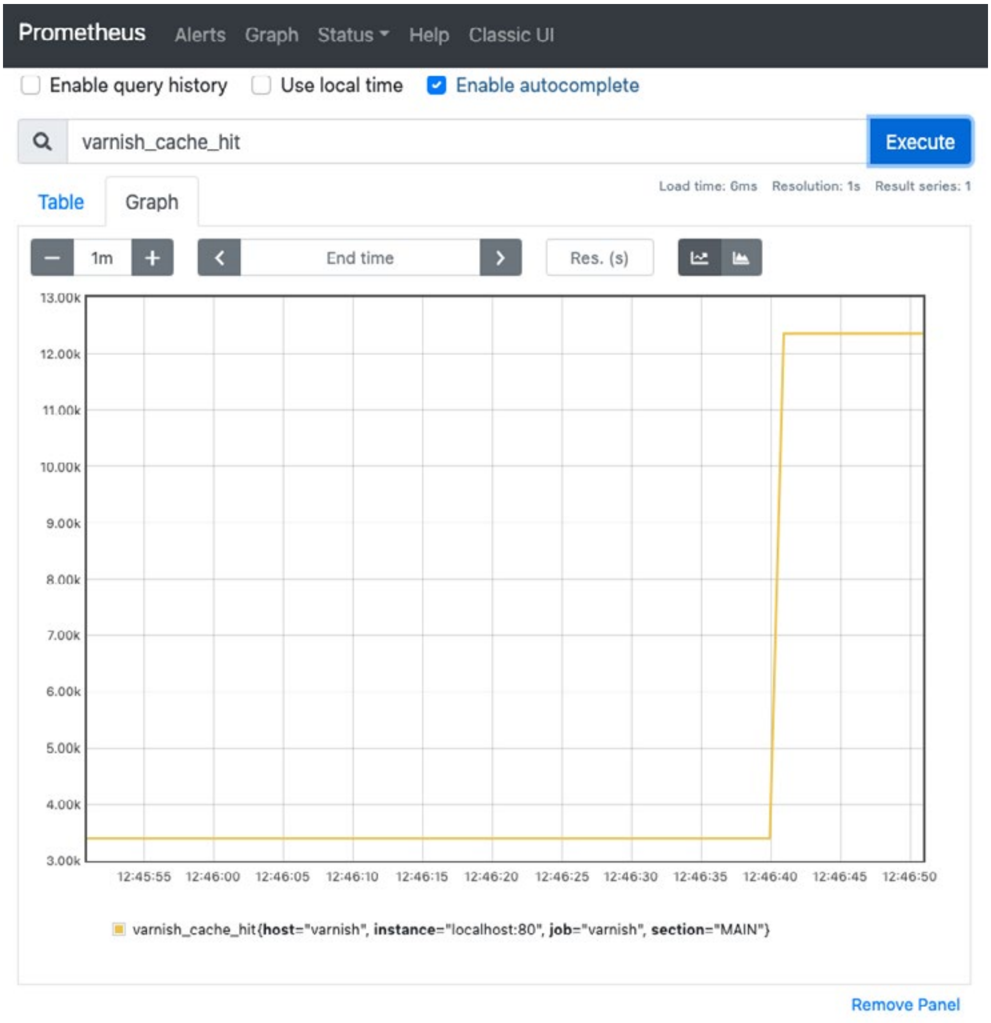
Setting up Prometheus

Now that we've figured out a way to export metrics from *Varnish*, we need to pull them from *Prometheus*. This requires a *Prometheus* server to be set up. Packages will do the job quite easily and come with a collection of exporters. The `/etc/prometheus/prometheus.yml` configuration file defines how metrics will be collected.

Here's a simplified configuration file that registers a *Varnish* exporter:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s
scrape_configs:
  - job_name: ‘varnish’
    static_configs:
    - targets: [‘varnish.example.com:9273’]
```

When you navigate to the homepage of your *Prometheus server*, you'll get a dashboard where you can search for metrics and where you can plot those metrics on a variety of graphs:



```
varnish_cache_hit
```

Load time: 6ms Resolution: 1s Result series: 1

varnish_cache_hit{host="varnish", instance="localhost:80", job="varnish", section="MAIN"}

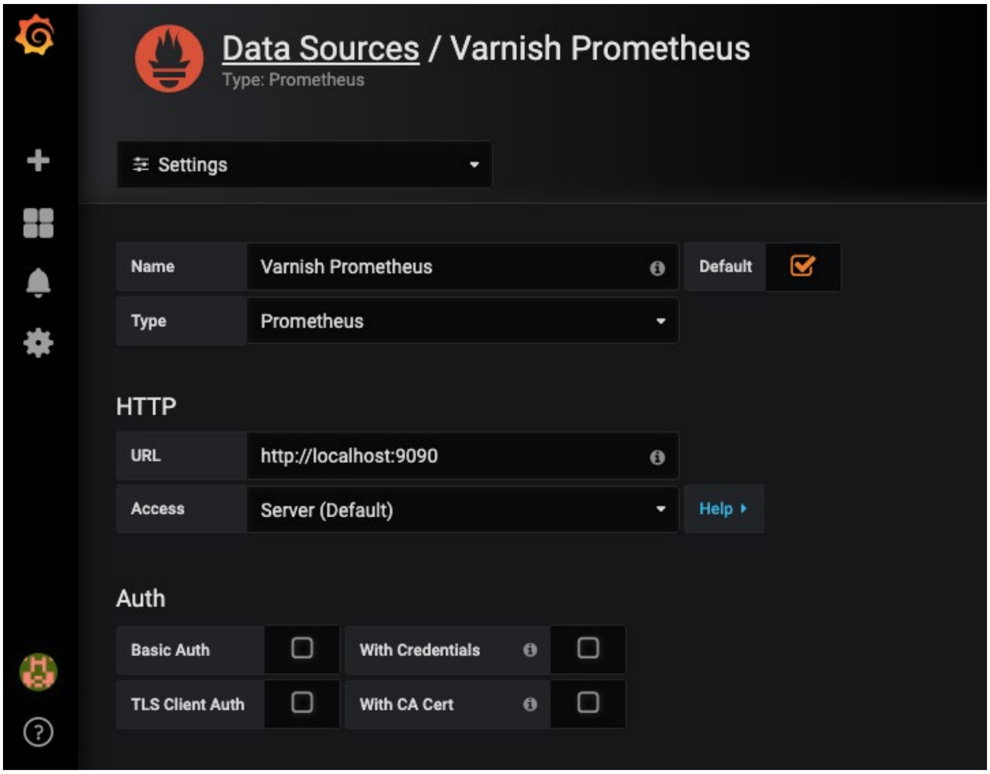
Prometheus

Although the *Prometheus* dashboard can do the job, it's not a real dashboard service. There are better alternatives, and *Grafana* is one of them.

Grafana

Grafana is a great way to visualize metrics and offers various different panels, dashboards, and plugins.

In our case, we want to make sure our *Prometheus* server is configured as a data source. This is very easy to configure in *Grafana* as you can see in the screenshot below:



Name: Varnish Prometheus
Default: ✓
Type: Prometheus

HTTP

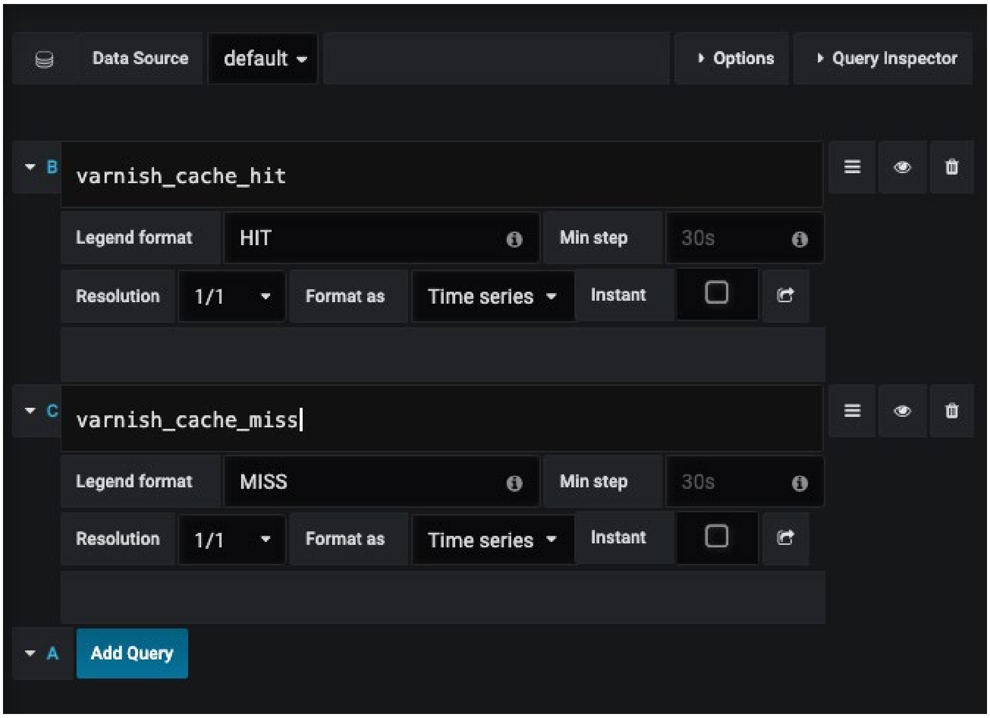
URL: http://localhost:9090
Access: Server (Default)

Auth

Basic Auth: ☐
With Credentials: ☐
TLS Client Auth: ☐
With CA Cert: ☐

Grafana data source

Once *Grafana* is aware of the *Prometheus* metrics, we can create a dashboard. As you can see in the screenshot below, you can plot multiple *Prometheus* metrics on a graph:


Data Source: default

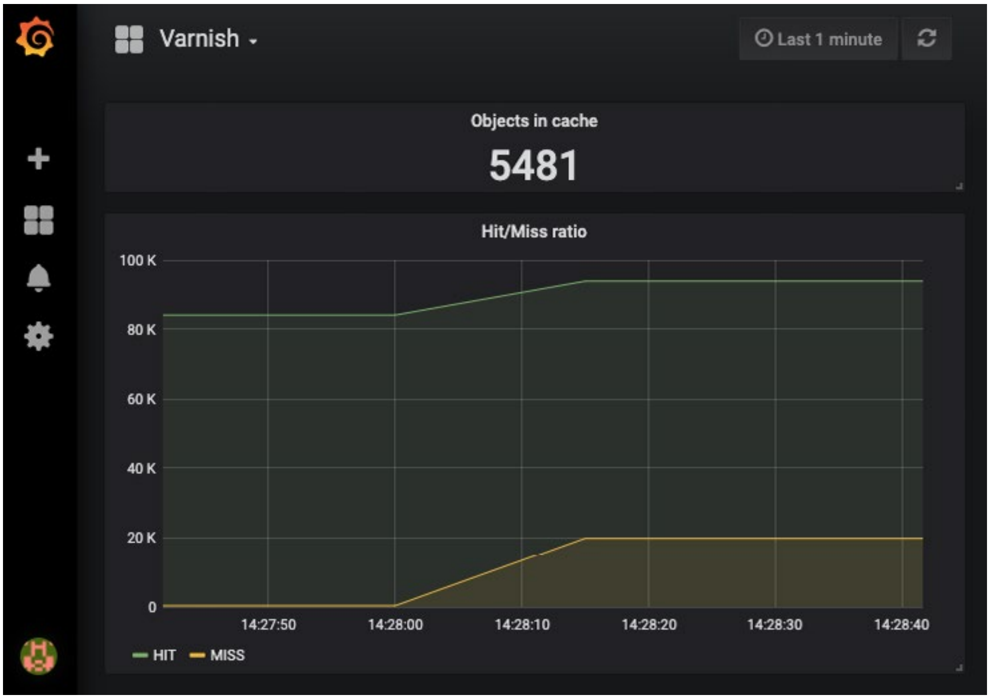
B: varnish_cache_hit
Legend format: HIT
Min step: 30s
Resolution: 1/1
Format as: Time series
Instant: ☐

C: varnish_cache_miss
Legend format: MISS
Min step: 30s
Resolution: 1/1
Format as: Time series
Instant: ☐

Grafana dashboard config

PromQL has various functions, expressions, and operators. You can filter metrics using various labels, and you can apply statistical functions. However, the details of this are beyond the scope of this book.

Here's what the result looks like:



Grafana

Besides the graph, we also defined a *Singlestat* that displays the current value of a metric. We use this to gauge the number of objects currently stored in cache.

7.7.4 Varnish Custom Statistics

We already know that *KVStore counters* can be used to display custom `varnishstat` metrics. However, there's also a *Varnish Enterprise* product called *Varnish Custom Statistics (VCS)*, which sends metrics from a *Varnish* server via the *VCS agent* to a *VCS server*.

The difference with `varnishstat` is that *VCS* performs measurements for a predefined set of metrics but grouped by *keys* that are defined in *VCL*.

VCS metrics

VCS metrics are stored in *timeseries buckets*. The length of a *bucket* defines the granularity of our measurements. The number of *buckets* we keep around will influence the total tracking time.

A bucket contains the following metrics:

Field	Description
`timestamp`	This is the timestamp for the start of the *bucket*'s period
`n_req`	The number of requests
`n_req_uniq`	The number of *unique* requests, if configured through the `vcs-unique-id` key in *VCL*
`n_miss`	Number of cache misses
`avg_restarts`	The average number of *VCL restarts* triggered per request
`n_bodybytes`	The total number of bytes transferred for the response bodies
`ttfb_miss`	Average *time to first byte* for requests that resulted in a backend request
`ttfb_hit`	Average *time to first byte* for requests that were served directly from cache
`resp_1xx` ... `resp_5xx`	Counters for response status codes
`reqbytes`	Number of bytes received from clients
`respbytes`	Number of bytes transmitted to clients
`berespbytes`	Number of bytes received from backends
`bereqbytes`	Number of bytes transmitted to backends
`pipe_bytes_in`	Number of bytes received from clients in pipe operations
`pipe_bytes_out`	Number of bytes transmitted to clients in pipe operations
`pipe_hdrbytes_req`	Number of bytes in headers received from clients where the request lead to a pipe operation
`pipe_hdrbytes_bereq`	Number of bytes in headers transmitted to backends where the request lead to a pipe operation

Based on these metrics, other metrics can be calculated. For example: by subtracting `n_miss` from `n_req`, we know the number of hits.

Defining keys

In your *VCL* file, you can define custom keys that will be collected by the *VCS agent* and aggregated in the *VCS server*.

Even if you don't specify any custom keys, the *VCS agent* will create three default keys if the `-d` option was passed to the `vcs-agent`:

- `ALL`
- `HOST/<host>`
- `URL/<url>`

The `ALL` key is a *static* key. It will be a constant value for all requests to Varnish. The `ALL` key essentially provides an overview of all incoming and outgoing requests.

The other two, `HOST/<host>` and `URL/<url>`, are dynamic keys. They reflect individual requests to Varnish. Dynamic keys provide insight into specific traffic patterns.

The *VCL* equivalent for these keys would be:

```
vcl 4.1;

import std;

sub vcl_deliver {
    std.log("vcs-key:ALL");
    std.log("vcs-key:HOST/" + req.http.Host);
    std.log("vcs-key:URL/" + req.http.Host + req.url);
}
```

For the `http://example.com/welcome` page, the keys would be as follows:

- `ALL`
- `HOST/example.com`
- `URL/example.com/welcome`

For each of these three keys the metrics can be retrieved via an *HTTP API*, *TCP output*, or the *graphical user interface*.

This is the *HTTP API* output of a single bucket for the `ALL` key:

```
{
    "timestamp": "2020-12-18T09:57:50+00",
    "n_req": 64,
    "n_req_uniq": 0,
    "n_miss": 11,
    "avg_restarts": 0.000000,
    "n_bodybytes": 72538778,
    "reqbytes": 12080,
    "respbytes": 72556553,
    "berespbytes": 9895,
    "bereqbytes": 2766,
    "pipe_bytes_in": 0,
    "pipe_bytes_out": 0,
    "pipe_hdrbytes_req": 0,
    "pipe_hdrbytes_bereq": 0,
    "ttfb_miss": 0.004838,
    "ttfb_hit": 0.000138,
    "resp_1xx": 0,
    "resp_2xx": 64,
    "resp_3xx": 0,
    "resp_4xx": 0,
    "resp_5xx": 0
}
```

Because our *VCS server* was configured with a *bucket length* of *two seconds*, we can conclude that *64 requests* were received in a *two-second time period*. There were *11 misses*. The logical conclusion is that there were *53 hits*. Every single request resulted in an `HTTP/2XX` response status code.

If we wanted to investigate the *11 misses*, we could request a *top miss* report from the *VCS server*:

```
{
    "ALL":11,
    "HOST/example.com":11,
    "URL/example.com/faq": 11
}
```

By default the *top miss* report will list up to ten keys that have the most misses in the latest bucket. We will talk more about *top reports* soon.

But what if you wanted more custom keys? As long as you log the right key, you can aggregate and query on any parameter that you want.

Let's say you want to see the metrics per *request method*. You'd use the following *VCL snippet* to make this happen:

```
std.log("vcs-key:METHOD/" + req.method);
```

This would result in keys like:

- `METHOD/GET`
- `METHOD/HEAD`
- `METHOD/POST`
- `METHOD/PUT`
- `METHOD/DELETE`
- `METHOD/OPTIONS`

Here's another interesting use case: imagine we wanted to measure the conversion from articles to a signup page. Basically, we're tracking conversions based on the `referer` request header.

Here's the *VCL* to get it done:

```
vcl 4.1;

import std;

sub vcl_deliver {
    if (req.url == "/signup" && req.http.referer ~ "^https?://[^\/]+/
article/([0-9]+)") {
        set req.http.articleid = regsub(req.http.referer,"^https?://
[^\/]+/article/([0-9]+)","\1");
        std.log("vcs-key:CONVERSION/SIGNUP/" + req.http.articleid);
        unset req.http.articleid;
    }
}
```

If for example a person is reading `https://example.com/article/1234` and clicks on a link that navigates to `/signup`, the resulting *VCS* key would be:

```
vcs-key:CONVERSION/SIGNUP/1234
```

And eventually, the *VCS server* will allow you to visualize, report, and query based on the custom `CONVERSION/SIGNUP/<articleid>` key.

The VCS agent

The `vcs-agent` program reads the *Varnish Shared Log* and captures `vcs-key` entries.

The agent then sends that data to the central `vcs` server program, which is aggregated and persisted.

The *VCS* components, including the `vcs-agent` program, are part of *Varnish Enterprise*. `vcs-agent` can be installed using the `varnish-custom-statistics-agent` package on your *Linux* system.

> The packages are proprietary and require a license key to be accessed.

If you're on *Debian* or *Ubuntu*, you can run the following command:

```
sudo apt-get install varnish-custom-statics-agent
```

And on *Red Hat* or *CentOS*, it's the following command:

```
sudo yum install varnish-custom-statistics-agent
```

The `vcs-agent` should be installed on every *Varnish* server you want to monitor. Once the package is installed, you have to configure the *Systemd service file* to make it work for you.

The following commands should be run to edit the configuration and restart the service with the new settings:

```
sudo systemctl edit --full vcs-agent
sudo systemctl restart vcs-agent
```

This is the standard `vcs-agent.service` file:

```
[Unit]
Description=Varnish Custom Statistics Agent
After=varnish.service

[Service]
Type=simple
# Give varnish a little startup pause
# Could be improved with notify functionality
ExecStartPre=/bin/sleep 2
ExecStart=/usr/sbin/vcs-agent -d localhost
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

The `-d` parameter is what creates the default `ALL`, `HOST`, and `URL` keys. The reference to `localhost` is the location of the *VCS server*. You'll most likely change that value and set it to the hostname of your *VCS server*.

There are other `vcs-agent` command line options you can tune:

- The `-g` option will output debug information. This could be useful when you're troubleshooting.
- The `-k` option allows you to customize the prefix that is used to match *VSL log entries*. The default value is `vcs-key`.
- The `-m` option sets the number of messages that should be queued when the *VCS* server is not available. Setting it to zero will create an unlimited queue.
- The `-p` option sets the remote port the agent should use when connecting to the server. The default value is `5558`.
- The `-n` option sets which *Varnish* instance the agent should read in case multiple *Varnish* instances are hosted on the same machine.

The VCS server

The *VCS server* collects, aggregates, and persists data from the *VCS agents*.

Installing the `vcs` can be done using packages.

The packages are also proprietary and require a license key to be accessed.

If you're on *Debian* or *Ubuntu*, you can run the following command:

```
sudo apt-get install varnish-custom-statics
```

And on *Red Hat* or *CentOS*, you use the following command:

```
sudo yum install varnish-custom-statistics
```

There's also a *Systemd unit file*. Here's the content of `vcs.service`:

```
[Unit]
Description=Varnish Custom Statistics
After=syslog.target network.target

[Service]
Type=forking
PIDFile=/var/run/vcs.pid
ExecStart=/usr/sbin/vcs \
    -P /var/run/vcs.pid \
    -Z

[Install]
WantedBy=multi-user.target
```

The `-P` option will set the location of the *PID* file, and the `-Z` option sets bucket time-stamps to use the UTC time zone.

If you want to modify the runtime configuration, you can run the following commands:

```
sudo systemctl edit --full vcs
sudo systemctl restart vcs
```

You can change some of the `vcs` options prior to restarting the service. Here are some of the options we haven't yet discussed:

- The `-b` option sets the length of an individual *bucket* in seconds. Defaults to 30. By setting `-b 2`, a bucket will aggregate two seconds worth of data. This can be considered the level of granularity for *VCS*.
- The `-m` option sets the maximum number of *buckets* for a key. Defaults to 15. If you multiply the number of buckets by the length of a bucket, you get the total duration of the tracking period.

- The `-z` option sets the listening port that a `vcs-agent` will connect to. Defaults to port `5558`.
- The `-a` option sets the listening port for the *HTTP interface*. Defaults to port `6555`.
- The `-l` option limits access to the *HTTP interface* using an *ACL*. Setting it to `-0.0.0.0/0,+192.168.0.0/16` will disable access for the general public, and only allow access from the `192.168.0.0` IP range.
- The `-O` option sets the *hostname* and *port* for *TCP output*. When set, *VCS* will transmit finished buckets in *JSON format* using TCP. This could be used for third-party processing or storage of the *VCS* data.
- The `-u` option sets the folder in which the *UI files* are stored. Defaults to `/usr/share/varnish-custom-statistics/html`.
- The `-g` option sets the *debug level*. This can be useful during troubleshooting.
- The `-K` option sets the credential file for *basic HTTP authentication* to protect access to the *HTTP endpoint*.
- The `-d` option sets the realm domain that is used with *basic HTTP authentication*.

The VCS API

Once you're up and running, and the `vcs-agent` is sending data, you can use the *HTTP API* to query *VCS* and to retrieve metrics in *JSON* format.

The *HTTP endpoint* runs on port `6555` by default and provides both the *API* and the *GUI*. There are various *HTTP resources* available at that endpoint.

The homepage, the `/#/live/keys` resource, and the `/#/live/realtime` resource display various elements of the *graphical user interface*, which we'll cover soon.

The `/key/<key>` resource, the `/match/<regex>` resource, the `/all` resource, and the `/status` are *API resources*.

Let's talk about `/all` first. This *URL* lists all available keys.

Here's some example output for `/all`:

```
{
    "keys": [
    "URL/example.com/",
    "URL/example.com/welcome",
    "URL/example.com/contact",
    "URL/www.example.com/",
    "ALL",
    "HOST/example.com",
    "HOST/www.example.com",
    "METHOD/GET",
    "METHOD/POST",
    ]

}
```

The output doesn't require a lot more explanation and should make sense if you paid attention when reading the *Defining keys* subsection.

To see the bucket metrics for all keys, just enable verbose mode `/all?verbose=1`:

```
{
    "keys": [
        "All": [
            {
              ...
            }
        ]
    ]
}
```

The `/match/<regex>` resource matches keys based on a *regular expression*. The regex should be *URL encoded*.

Here's an example where we match all keys that start with `URL/example.com`:

```
/match/URL%2Fexample.com%2F
```

This could be the output:

```
{
    "keys": [
    "URL/example.com/",
    "URL/example.com/welcome",
    "URL/example.com/contact"
    ]
}
```

Another way of listing and sorting keys is with a *top report*.

A *top report* will examine all the keys in a request and return a sorted list of keys that is based on a specific metric.

The keys that will be examined by a *top report* is determined by either the `/all` or the `/match/<regex>` *API URL*.

By appending one of the *top reports* to your key listing request, a sorted list of keys will be returned:

- `/top` Sort based on number of n_req metric
- `/top_ttfb` Sort based on the ttfb_miss metric
- `/top_size` Sort based on the n_bodybytes metric
- `/top_miss` Sort based on the n_miss metric
- `/top_respbytes` Sort based on the respbytes metric
- `/top_reqbytes` Sort based on reqbytes metric
- `/top_berespbytes` Sort based on berespbytes metric
- `/top_bereqbytes` Sort based on bereqbytes metric
- `/top_restarts` Sort based on the avg_restarts metric
- `/top_5xx`, `/top_4xx`, `/top_3xx`, `/top_2xx`, `/top_1xx` Sort based on number of resp_xxx metric
- `/top_uniq` Sort based on the n_req_uniq metric

Here are a couple of *top report* examples:

```
/all/top
/all/top_4xx
/match/URL%2Fexample.com%2F/top_miss
/match/URL%2Fexample.com%2F/top_size
```

By default a *top report* will be ranked by a key's latest bucket and will return a list of ten sorted keys.

It is possible to customize the *top report* by specifying the bucket `?b=<buckets>` query parameter and/or the count `/<count>` parameter in the *API URL*.

The count parameter determines the number of keys that will be returned.

Imagine if we wanted the top three keys that have the most cache misses. This would be the *API URL*:

```
/all/top_miss/3
```

If we wanted a similar report for all keys matching the `example.com` URL, this would be the *API URL*:

```
/match/URL%2Fexample.com/top_miss/3
```

When processing a *top report*, the ranking will be based on a metric value in one or more *buckets*. By default only the key's latest bucket will be used, but the bucket `?b=<buckets>` query parameter can specify the total number of buckets that should be summarized for a key.

Here's an example where we want the top eight keys that have the most misses for the last five buckets:

```
/all/top_miss/8/?b=5
```

Once you've figured out what key you want to examine, based on the various key filtering techniques, you can use the `/key/<key>` *API URL* to retrieve all the key's metrics.

Here's the *API URL* we're accessing when we want the metrics for the `URL/example.com/welcome` key:

```
/key/URL%2Fexample.com%2Fwelcome
```

This could be the output:

```
{
    "URL/example.com/welcome": [
        {
            "timestamp": "2020-12-18T14:05:34+00",
            "n_req": 877,
            "n_req_uniq": 0,
            "n_miss": 0,
            "avg_restarts": 0.000000,
            "n_bodybytes": 546371,
            "reqbytes": 149967,
            "respbytes": 788865,
            "berespbytes": 0,
            "bereqbytes": 0,
            "pipe_bytes_in": 0,
            "pipe_bytes_out": 0,
            "pipe_hdrbytes_req": 0,
            "pipe_hdrbytes_bereq": 0,
            "ttfb_miss": 0,
            "ttfb_hit": 0.000129,
            "resp_1xx": 0,
            "resp_2xx": 877,
            "resp_3xx": 0,
            "resp_4xx": 0,
            "resp_5xx": 0
        },
        {
            "timestamp": "2020-12-18T14:05:32+00",
            "n_req": 249,
            "n_req_uniq": 0,
            "n_miss": 1,
            "avg_restarts": 0.000000,
            "n_bodybytes": 155127,
            "reqbytes": 42579,
            "respbytes": 223995,
            "berespbytes": 814,
            "bereqbytes": 193,
            "pipe_bytes_in": 0,
            "pipe_bytes_out": 0,
            "pipe_hdrbytes_req": 0,
            "pipe_hdrbytes_bereq": 0,
            "ttfb_miss": 0.006705,
            "ttfb_hit": 0.000286,
            "resp_1xx": 0,
            "resp_2xx": 249,
            "resp_3xx": 0,
            "resp_4xx": 0,
            "resp_5xx": 0
        }
    ]
}
```

And finally, there's the `/status` resource that displays some simple counters which represent the state of the *VCS server:*

```
{
    "uptime": 93033,
    "n_keys": 24,
    "n_trans": 682026,
    "db_mem_usage": 1197760,
    "output_queue_len": 0
}
```

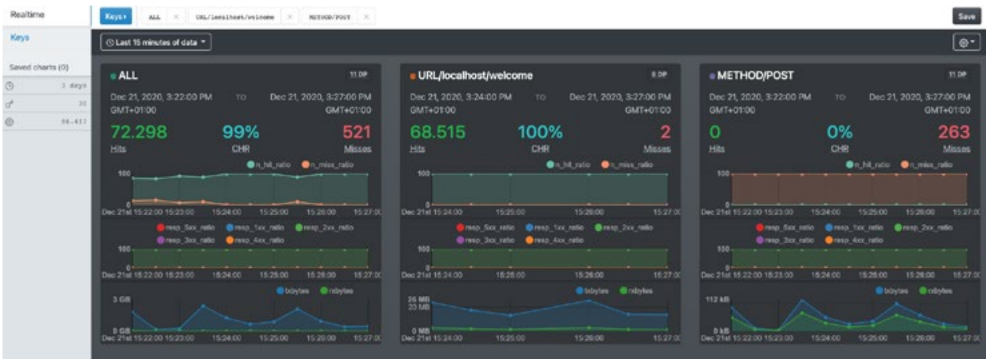
Let's break down this output:

- The server has been up and running for *93033 seconds*.
- We're currently tracking *24 keys*.
- We've processed *682026 transactions*.
- The storage engine is estimated to consume *1197760 bytes* of memory.
- There is currently no output queue.

The VCS user interface

The *HTTP API* returns a lot of useful information. There is surely enough flexibility to sort, group, and filter the metrics. However, the output isn't intuitive. That's why the `vcs` *HTTP endpoint* can also return a *graphical user interface*.

This *GUI* leverages the *API* and visualizes the metrics through *graphs* and *counters*. We'll illustrate this fact using a set of screenshots.

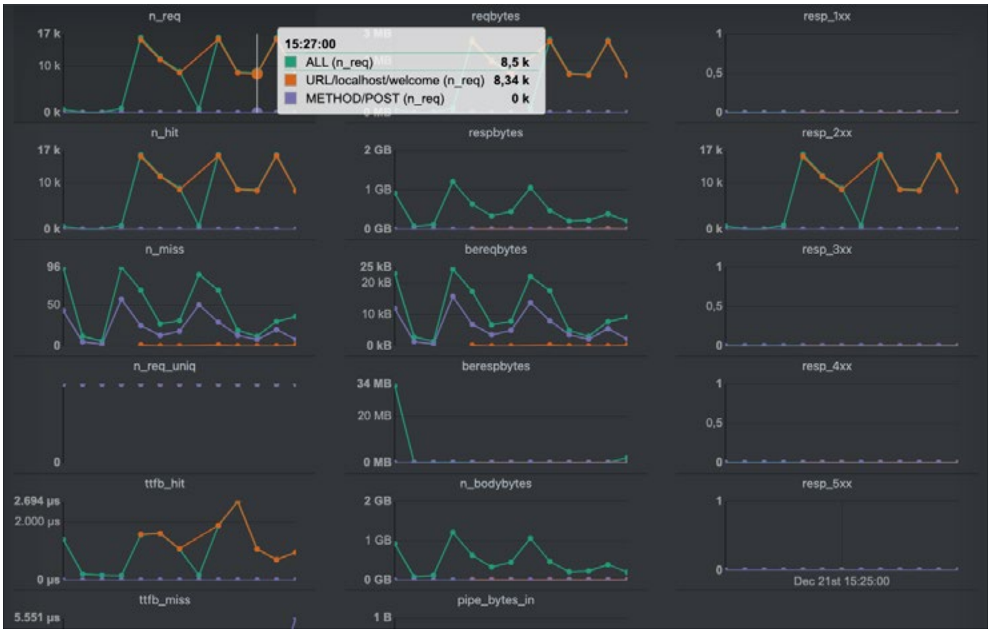


VCS metrics per key

This first screenshot displays a set of metrics for three selected keys:

- `ALL`
- `URL/localhost/welcome`
- `METHOD/POST`

The metrics for the `ALL` key indicate that we have a *cache hit ratio (CHR)* of 99%. All responses were in the *2XX* range. When we look at the results of the `URL/localhost/welcome` key, we notice a nearly 100% hit rate. However, the `METHOD/POST` keys as a 0% hit rate, which makes sense given *built-in VCL* behavior.

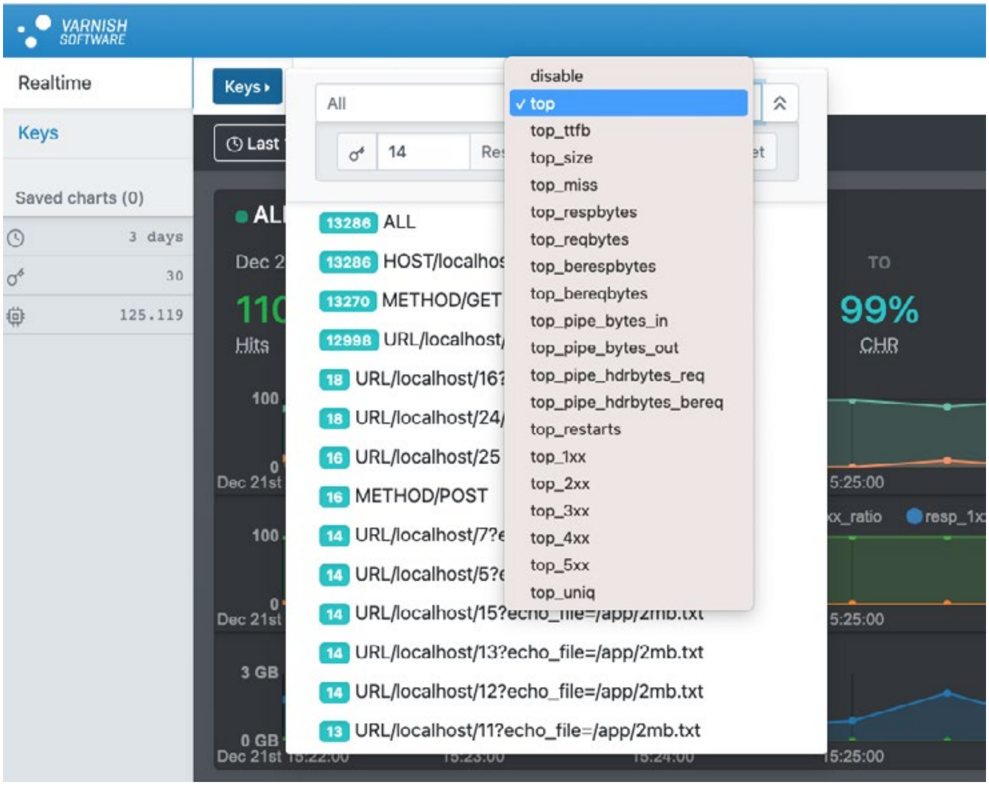


VCS metrics

When we scroll down, we see the summarized metrics for all keys, which are shown in the second screenshot.

Each metric is displayed separately, and the results are plotted on the graph per selected key. We highlighted a single measurement point on the `n_req` graph, which visualizes the number of requests.

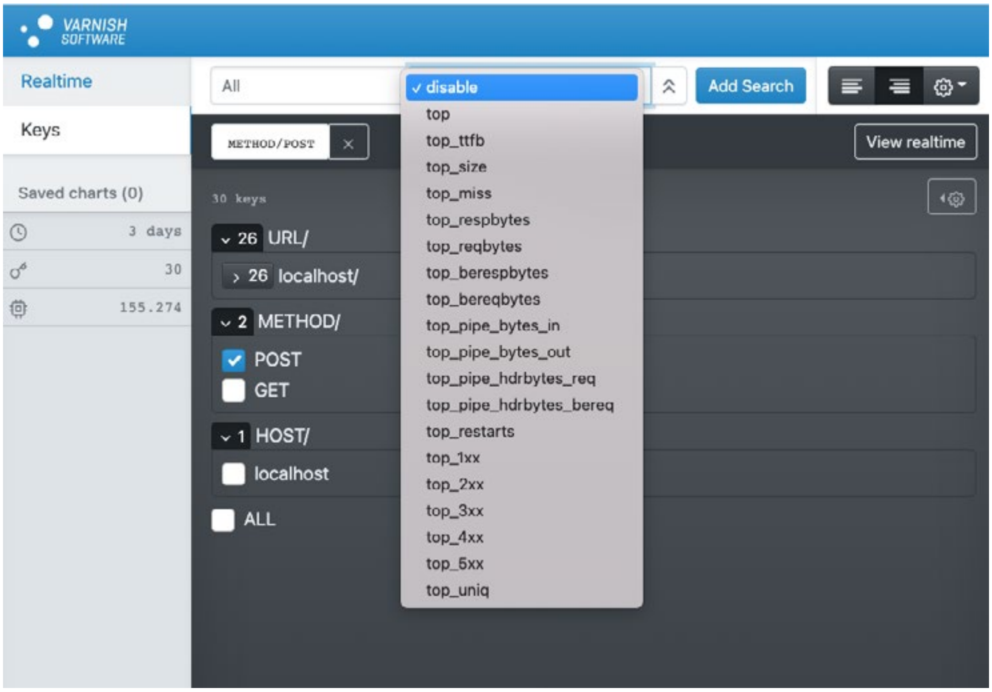
Our setup had a *bucket length of 30 seconds*, which means that in this 30-second interval, *8500 requests* were processed, 8430 of which were sent to `http://localhost/welcome`. The amount of *HTTP POST* requests wasn't high enough to be visible in this interval.



VCS key filer

The following screenshot shows what happens when you press the keys button. It allows you to select keys based on a search box. Top keys can be sorted on specific metrics, and the count and bucket parameters can be specified.

The list shows how metrics are available for the matching keys. By selecting a key, the corresponding metrics are added to the overview.



VCS key explorer

The final screenshot features a separate part of the *VCS GUI* that focuses on key selection. Key search and sorting is also supported. The selected keys can then be turned into a dashboard by pressing the *View realtime* button.

7.7.5 When things go wrong

Metrics are interesting: graphs are fun and fancy, but they serve a purpose. The main purpose is understanding what's going on under the hood. And when something goes wrong, the metrics and graphs should help you investigate why.

It's not just about uptime and availability. Classic monitoring systems that perform an *HTTP check* can easily spot when *Varnish* is down. The problems that occur are often subtler and more related to performance and cacheability.

With the dozens of metrics that are available, it's tough to see the forest for the trees. And a basic metric like the *hit rate* isn't really a tell-tale sign.

The absolute values of the counters are often irrelevant: it's the change rate that matters. However there are still some metrics you want to keep as low as possible.

Counters we want as low as possible

The `MGT.child_panic` is one of the counters we want to see as low as possible. A panic is always something that should be avoided. Panics can be displayed by running `varnishadm panic.show`.

The `MAIN.thread_queue_len` metric should also be kept as low as possible. Queued threads indicate a lack of available threads, which means your `varnishd` process is either super busy, or your `thread_pool_max` runtime parameter is set too low.

The `MAIN.sess_fail` metric should also be as low as possible. This counter is the sum of all other `MAIN.sess_fail_*` counters. Here's a list of counters that track session failures:

- `MAIN.sess_fail_econnaborted`: connection aborted by the client, which is usually quite harmless.
- `MAIN.sess_fail_eintr`: the `accept()` system call was interrupted.
- `MAIN.sess_fail_emfile`: there are not file descriptors available.
- `MAIN.sess_fail_ebadf`: the listen socket file descriptor was invalid.
- `MAIN.sess_fail_enomem`: insufficient socket buffer memory.
- `MAIN.sess_fail_other`: some other reason, which will be clarified in the debug log.

It's not just the session failures that indicate problematic behavior. Sessions could also be queued while waiting for an available thread. The `MAIN.sess_queued` counter exposes this kind of behavior. The `MAIN.sess_dropped` counter will increase when there is a lack of worker threads, and when the session queue is too long. The `thread_queue_limit` runtime parameter controls the *size of the thread queue per thread pool*.

Dropped sessions happen for *HTTP/1.1* traffic. For *HTTP/2* traffic, the `MAIN.req_dropped` counter is used to count dropped streams.

The `MAIN.n_lru_nuked` counter indicates that the cache was full and that the *least recently used* object was removed to free up space. When this counter skyrockets, you should consider increasing the size of your cache or re-evaluate the *TTLs* of your objects.

If you're using *MSE on Varnish Enterprise*, you should have a look at `MSE.*.n_lru_nuked` if you want to monitor object nuking.

Nuked objects aren't great, but they are not catastrophic. But if the `MAIN.n_lru_limited` counter increases, `varnishd` wasn't able to nuke enough space to fit the new object. The corresponding fetch will fail, and the end user will either receive no response, or a partial response, because of content streaming.

The `nuke_limit` runtime parameter indicates the number of nuking attempts. The standard value is 50, and after 50 nuking attempts, *Varnish* will give up, and the `MAIN.n_lru_limited` counter will increase.

> The failure to create enough free space is often a race condition between the thread that clears the space and another thread that tries to insert a new object. As mentioned before, *MSE* has a level of isolation to avoid these issues.

Debugging

There are many other reasons why things go wrong, and many counters to visualize these issues. However, the counters tell you what's going on, but not why these issues are occurring.

We have to revert to *Varnish Shared Memory Logging* to get more information, and the `Debug` tag can help us with that. We can either include the `Debug` tag in your regular *VSL output*, but we can also filter out debugging information using the following command:

```
varnishlog -g raw -I Debug
```

The *raw transaction grouping* will ensure debug information is not restricted to sessions and requests. But as mentioned before, without *VSL queries* or *rate limiting*, the output will become overwhelming on a busy production server.

By default *Varnish* will not keep track of debugging information, but the `debug` runtime parameter can enable and disable specific debugging features.

There is an extensive list of debugging features, some of which are only used for testing. Here's an example that includes some useful debugging symbols, which will be exposed in *VSL*:

```
varnishd -p debug=+req_state,+workspace,+waiter,+waitinglist,+lurk-
er,+processors,+protocol
```

But as mentioned, enabling them all will result in a lot of *noise*. Be selective, consider using *VSL queries*, and also consider using *rate limiting*.

7.7.6 Varnish scoreboard

In *chapter 1*, we talked about the *Varnish threading model* as part of the *Under the hood* section.

It was made clear that both the *manager* and the *child* process use threads to perform various tasks.

`varnishstat` allows us to monitor various metrics, some of which relate to *Varnish's threads*. `varnishlog` provides information about individual actions that leverage threads behind the scenes.

By running `varnishscoreboard` you can actually monitor the state of the currently active threads on your *Varnish* server. This utility is part of *Varnish Enterprise* and was redesigned for *Varnish Enterprise 6.0.7r1*.

Here's some example output:

```
$ varnishscoreboard
Age      Type     State         Transaction Parent      Address
Description
   1.64s probe    waiting                 0           0 -
boot.default
   2.11m acceptor accept                  0           0 :6443
a1
   2.11m acceptor accept                  0           0 :6443
a1
   2.11m acceptor accept                  0           0 :80
a0
   0.03s acceptor accept                  0           0 :80
a0
   0.01s backend  startfetch         360910      360909 -
POST example.com /login
   0.01s client   fetch              360909      360908
172.19.0.1:63610    POST example.com /login
   2.11m acceptor accept                  0           0 :6443
a1
   2.11m acceptor accept                  0           0 :6443
a1
   2.11m acceptor accept                  0           0 :80
a0
   0.01s acceptor accept                  0           0 :80
a0
Threads running/in pool: 10/90
```

This output indicates that we have *100 worker threads available*. Ten are running, and 90 are waiting in the thread pool. As we learned in the *Under the hood* section, there are a fixed number of threads for various tasks, but the worker threads are dynamic and are kept in thread pools.

The thread number in this example reflects the dynamic threads. The static ones are not reflected.

We also learn that there is a health probe available, which monitors `boot.default`. This refers to the `default` backend, which was defined in the `boot` *VCL configuration*.

There are also some acceptor threads active, both on *port 6643* for TLS and *port 80* for plain HTTP. These threads are responsible for accepting new connections.

There is a client-side thread that is fetching content from the backend. This is done for an *HTTP POST* request to `http://example.com/login`. This is also reflected in a corresponding thread that performs a backend fetch.

Here's another example:

```
$ varnishscoreboard
Age       Type     State         Transaction Parent      Address
Description
   0.00s session  newreq           11436541          0
172.19.0.1:60542
   1.07s client   transmit          8356616     8356615
172.19.0.1:60384    GET localhost /download/video.mp4
   0.37s client   transmit         11370849     7078552 -
GET localhost /download/audio.mp3
   0.00s acceptor accept                  0           0 :80
a0
   3.22s probe    waiting                 0           0 -
boot.default
  37.70m acceptor accept                  0           0 :6443
a1
  37.70m acceptor accept                  0           0 :6443
a1
  37.70m acceptor accept                  0           0 :80
a0
   0.00s acceptor accept                  0           0 :80
a0
  37.70m acceptor accept                  0           0 :6443
a1
  37.70m acceptor accept                  0           0 :6443
a1
  37.70m acceptor accept                  0           0 :80
a0
Threads running/in pool: 11/89
```

In this example you can also see new requests being made in the form of *session threads*. And you can also witness *client threads* performing transmissions. For large files it is easier to spot transmissions, as they take more time.

By default the `varnishscoreboard` doesn't output anything. By enabling the `scoreboard_enable` runtime parameter in `varnishd`, the scoreboard will emit the right data. Please note that as of *Varnish Enterprise 6.0.7r1* the `scoreboard_enable` runtime parameter is a deprecated alias of `thread_pool_track`.

7.8 Logging

When you put *Varnish* in front of your *origin* servers, the logs on the *origin* will not be very useful. Because the goal is to dramatically reduce the number of requests to the *origin*, the server logs will only contain a fraction of what the actual traffic represents.

That's why *Varnish* has a quite extensive logging mechanism: not only to mimic *NCSA-style* logs, but to give operators insight about the flow within *Varnish*.

Varnish comes with various logging tools to provide this insight:

- `varnishlog`: displays log entries
- `varnishtop`: presents a continuously updated list of the most commonly occurring log entries
- `varnishncsa`: formats log entries in *Apache/NCSA "combined" log format*

7.8.1 Varnish Shared Memory Log

Storing *Varnish* logs in files by default is a bad idea. A system like *Varnish* is used to process massive amounts of concurrent requests. If every request were to be persisted on disk as a log line, the load on the system would be unbearable.

Even if your system can handle the load, there's the challenge of fitting these logs on disk. Depending on the kind of logs you generate, the verbosity and size can be huge.

Although it is possible to have log files in *Varnish*, the standard mechanism is to store logs in an in-memory circular buffer. This means that as soon as the buffer is full, it is overwritten.

We call this the *Varnish Shared Memory Log (VSL)*, and it is the core of *Varnish*'s logging infrastructure. The `vsl_space` runtime parameter defines the size of the *VSL*, which defaults to *80 MB*. This value can be increased all the way up to *4 GB*.

> The `-l` option of `varnishd` is shorthand for `-p vsl_space=`

`varnishlog`, `varnishtop`, and `varnishncsa` use the *VSL* as their source of input and offer various ways to format, filter, and query the logs.

Here's a quick teaser of what `varnishlog` output looks like:

```
*   << Request  >> 2
-   Begin          req 1 rxreq
-   Timestamp      Start: 1606224128.870382 0.000000 0.000000
-   Timestamp      Req: 1606224128.870382 0.000000 0.000000
-   ReqStart       172.19.0.1 37632 a0
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   ReqHeader      User-Agent: curl/7.64.1
-   ReqHeader      Accept: */*
-   ReqHeader      X-Forwarded-For: 172.19.0.1
```

This output is only an extract because the full log output for this transaction is far too verbose.

7.8.2 Transactions

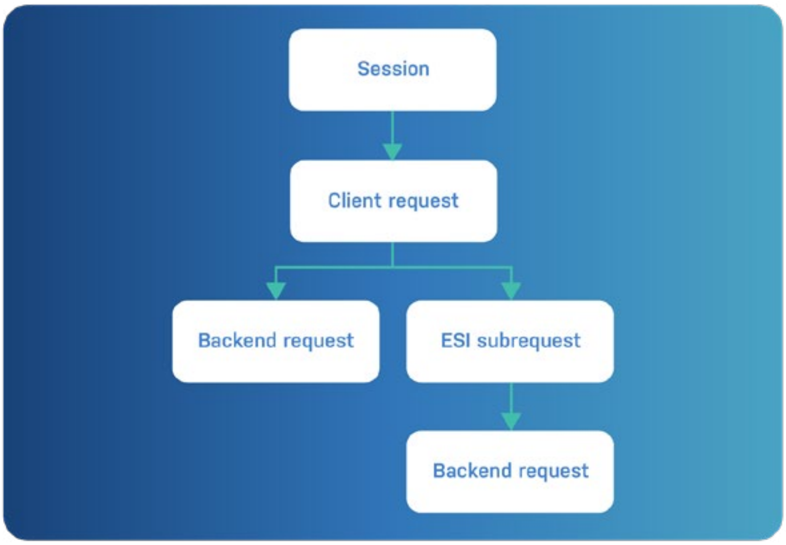
VSL keeps track of transactions. Each transaction contains a number of log lines and is identified by a *transaction identifier*, which we call the *VXID*.

Transactions are either *sessions* or *requests*:

- A *session* represents the *TCP* connection between the *client* and *Varnish*.
- A *request* is any *HTTP* request that involves *Varnish*.
- There are different kinds of *request transactions*:
- The *client request* to *Varnish*
- The *backend request* from *Varnish* to the *origin*
- An *ESI subrequest* from *Varnish* to the *origin*

Transaction hierarchy

There is a hierarchy between transactions, which is illustrated in the diagram below:



VSL transaction hierarchy

- The *session* is the top-level transaction.
- A session can have multiple *requests*.
- Non-cached objects will result in *backend requests*.
- Backend responses may include *ESI tags*. When parsed, they result in *ESI subrequests*.
- Non-cached *ESI* responses may result in corresponding *backend requests*.

Here's what this looks like in *VSL*:

```
*   << BeReq    >> 3
-   Begin          bereq 2 fetch

*   << BeReq    >> 5
-   Begin          bereq 4 fetch

*   << Request  >> 4
-   Begin          req 2 esi

*   << Request  >> 2
-   Begin          req 1 rxreq

*   << Session  >> 1
-   Begin          sess 0 HTTP/1
```

The `Begin` tag is used at the start of each transaction.

The first `<< BeReq >>` transaction has *VXID 3*. The `Begin` tag for this transaction indicates that this is a regular fetch that was initiated by *VXID 2*.

The second `<< BeReq >>` transaction has *VXID 5*. The `Begin` tag for this transaction indicates that this is a regular fetch that was initiated by *VXID 4*.

The first `<< Request >>` transaction is identified by *VXID 4*, and its `Begin` tag is an ESI subrequest that was initiated by *VXID 2*.

The second `<< Request >>` transaction is identified by *VXID 2*, and its `Begin` tag is a regular request that was initiated by *VXID 1*.

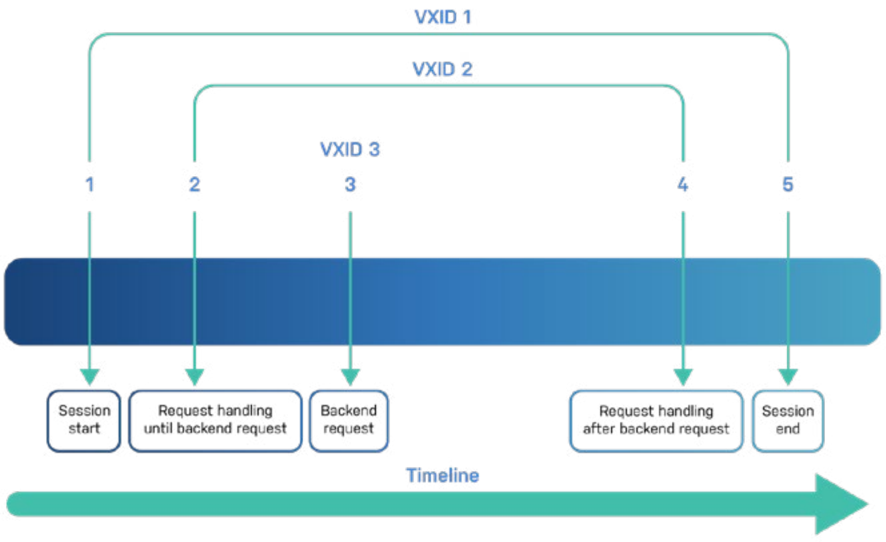
The `<< Session >>` transaction is identified by *VXID 1* and opens up a connection for `HTTP/1.1` traffic. It doesn't depend on any other transaction, hence the reference to *VXID 0*.

Transaction grouping

Transactions can be grouped, and the type of grouping that is used will influence the order of the transactions.

By default transactions are displayed in the order in which they complete, which at first glance looks like the reverse order and can seem puzzling.

Here's another diagram that should illustrate this effect:


VXID 1
VXID 2
VXID 3
1
2
3
4
5
Session start
Request handling until backend request
Backend request
Request handling after backend request
Session end
Timeline

VSL transaction hierarchy timeline

In this simple example, the *backend request*, identified by *VXID 3*, is the only transaction that doesn't depend on another and is displayed first.

The *request*, identified by *VXID 2*, can only complete after the *backend transaction* and will be displayed next.

And finally the *session*, identified by *VXID 1*, is displayed because it was waiting for the *client request* to complete.

You can choose the following grouping modes to change the order:

- vxid (default)
- session
- request
- raw

This is our *VSL output* grouped by *session*:

```
*   << Session  >> 1
-   Begin          sess 0 HTTP/1
**  << Request  >> 2
--  Begin          req 1 rxreq
*** << BeReq    >> 3
--- Begin          bereq 2 fetch
*** << Request  >> 4
--- Begin          req 2 esi
*4* << BeReq    >> 5
-4- Begin          bereq 4 fetch
```

As you can see, the order is more intuitive, and there's a level of indentation.

In situations where you primarily care about requests and not the TCP session information, you can group by *request* and omit the session transaction.

Here's what that looks like:

```
*   << Request  >> 2
-   Begin          req 1 rxreq
**  << BeReq    >> 3
--  Begin          bereq 2 fetch
**  << Request  >> 4
--  Begin          req 2 esi
*** << BeReq    >> 5
--- Begin          bereq 4 fetch
```

You can see that *VXID 1* is no longer included and that *VXID 2* is not the top-level transaction. And the indentation remained.

As far as *raw grouping* goes, here's some example output:

```
   1 Begin          c sess 0 HTTP/1
   3 Begin          b bereq 2 fetch
   5 Begin          b bereq 4 fetch
   4 Begin          c req 2 esi
   2 Begin          c req 1 rxreq
```

As you can see, the log lines are no longer grouped in transactions. They are displayed as they are received. As soon as some production traffic hits your *Varnish* server, *raw* grouping becomes nearly impossible to use.

However, *raw grouping* still has its purpose and is also the only way to collect non-transactional logs, logs that aren't tied to a *session*, *request* or *backend request* task. Non-transactional logs have the *VXID 0* to reflect the absence of a transaction ID.

When using `varnishlog`, `varnishtop`, or `varnishncsa`, the `-g` parameter can be used to control the grouping.

Here are some examples:

```
varnishlog -g request
varnishlog -g session
varnishlog -g vxid
varnishlog -g raw
```

7.8.3 Tags

Each *VSL transaction* contains a number of log lines. Each line has a *tag* and that tag has a corresponding value.

In the previous examples we've limited ourselves to the `Begin` tag to indicate the beginning of each transaction. But there are a lot more tags that can be displayed in the *VSL output*.

Transaction tags

We first need to finish our work with transactions. Here's the list of transaction-related tags:

- `Begin`: marks the start of an *VXID transaction*
- `End`: marks the end of an *VXID transaction*
- `Link`: links to a child *VXID transaction*

And here's an example where these tags are used:

```
*   << Request  >> 2
-   Begin          req 1 rxreq
-   Link           bereq 3 fetch
-   End
**  << BeReq    >> 3
--  Begin          bereq 2 fetch
--  End
```

This is the format of the `Begin` tag:

```
%s %d %s
|  |  |
|  |  +- Reason
|  +---- Parent vxid
+------- Type ("sess", "req" or "bereq")
```

As you can see it starts either `sess`, `req`, or `bereq` to indicate the type of transaction. The parent *VXID* has been covered extensively, so there's no reason to elaborate on that. The *reason* field clearly indicates why or how this transaction is taking place.

The `End` tag may look a bit unnecessary, because it seemingly doesn't add any value to the logs. But it is used internally to delimit transactions, and if a transaction takes too long to complete an `End synth` record might be synthesized to reflect that. The `varnishlog` options `-L` and `-T` might lead to incomplete transactions being displayed.

The `Link` tag is also quite useful, as it shows what kind of *child transaction* is initiated from the current one.

Here's the format:

```
%s %d %s
|  |  |
|  |  +- Reason
|  +---- Child vxid
+------- Child type ("req" or "bereq")
```

A *transaction* can either trigger a *(sub-)request*, or a *backend request*. Hence the types `req` and `bereq`. The `Link` tag will also display the *VXID* of the child transaction. And finally, the reason why this happens is displayed.

Session tags

- `SessOpen`: displays the *socket endpoints* when a client connection has been opened
- `SessClose`: displays the reason for the socket closure and the duration of the session

Here's an example where these two tags are used within a *session transaction*:

```
*   << Session   >> 1
-   SessOpen        127.0.0.1 51726 a0 127.0.0.1 80 1606299467.161264
19
-   SessClose       REM_CLOSE 0.004
```

In order to understand what the values of `SessOpen` are, here's the format for this tag:

```

%s %d %s %s %s %f %d
|  |  |  |  |  |   |
|  |  |  |  |  |   +- File descriptor number
|  |  |  |  |  +----- Session start time (unix epoch)
|  |  |  |  +-------- Local TCP port
|  |  |  +----------- Local IPv4/6 address
|  |  +-------------- Socket name (from -a argument)
|  +----------------- Remote TCP port
+-------------------- Remote IPv4/6 address
```

So let's explain the meaning of the values from the example:

- `127.0.0.1` is the *IP address* of the client.
- `51726` is the *port number* of the client.
- `a0` is the name of our socket. Because we didn't name it via `-a`, the name `a0` was automatically assigned by *Varnish*.
- `127.0.0.1` is also the *IP address* of the server.
- `80` is the *port number* of the server. This indicates regular *HTTP* traffic.
- `1606299467.161264` is the *epoch* equivalent of `Wednesday, November 25, 2020 10:17:47.161 AM`.
- `19` is the file descriptor number.

And from the `SessClose` tag we can determine that the session was closed normally and lasted *four milliseconds*.

Request tags

The most commonly used *VSL tags* are request tags. They refer to the *HTTP request* that is made by the client.

Here's an overview of these tags:

- `ReqStart`: the start of request processing
- `ReqMethod`: the HTTP method used to perform the request
- `ReqURL`: the URL that is addressed by the request
- `ReqProtocol`: the HTTP version used by the request
- `ReqHeader`: a set of request headers that were passed
- `ReqAcct`: the byte counts for the request handling

Here's an example that features these tags:

```
-   ReqStart       127.0.0.1 51726 a0
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   ReqHeader      User-Agent: curl/7.64.0
-   ReqHeader      Accept: */*
-   ReqHeader      X-Forwarded-For: 127.0.0.1
-   ReqAcct        73 0 73 268 540 808
```

The `ReqStart` tag lets us know what the IP address and port number of the client is. The `a0` refers to the socket that was used to connect. This socket is defined using the `-a` startup parameter.

The `ReqMethod` tag indicates a regular *HTTP* `GET` request, and the `ReqURL` shows that this request happened on the `/` URL.

The *HTTP protocol version* for this request was `HTTP/1.1`, as indicated by the `ReqProtocol` tag.

A set of `ReqHeader` tags are used to list the request headers that were passed with this *HTTP request*.

Finally, the `ReqAcct` tag is there to provide *request accounting* information. Here's the format of this tag, which will allow us to explain the meaning of `ReqAcct` from our example:

```
%d %d %d %d %d %d
|  |  |  |  |  |
|  |  |  |  |  +- Total bytes transmitted
|  |  |  |  +---- Body bytes transmitted
|  |  |  +------- Header bytes transmitted
|  |  +---------- Total bytes received
|  +------------- Body bytes received
+---------------- Header bytes received
```

What this means in our example is that the incoming request contained *73 bytes of request headers*, and no request body, which results in a total incoming byte count of *73* bytes.

The response contained *268 bytes of response header data*, and *540 bytes* of body payload. This body payload size will also be reflected in the `Content-Length` response header.

The total number of transmitted bytes is *808*.

Response tags

Whenever there's an *HTTP request*, there must an *HTTP response*. The *response tags* are responsible for displaying information about the *HTTP response* that was generated by *Varnish*.

Here's the list of *response tags*:

- `RespProtocol`: the HTTP version that is used in the response
- `RespStatus`: the HTTP response status code of the response
- `RespReason`: the *response-reason phrase* that clarifies the status
- `RespHeader`: a set of response headers that were returned

Here's a short example of a response containing the *response tags* that were mentioned:

```
-   RespProtocol   HTTP/1.1
-   RespStatus     200
-   RespReason     OK
-   RespHeader     Content-Type: text/html
-   RespHeader     Content-Encoding: gzip
-   RespHeader     X-Varnish: 2
-   RespHeader     Age: 1000
```

We know the response was made using the `HTTP/1.1` protocol, as indicated in by the `RespProtocol` tag. The response was a regular `200 OK`. The `RespStatus` and `RespReason` will return this information. And finally, there is a set of `RespHeader` tags that contains the *HTTP response headers* that were returned.

In this case two response headers were injected by *Varnish*:

- `Age`: how long the object has been in cache
- `X-Varnish`: the *VXID* of the transaction

Backend tags

The *backend tags* are used when a *miss or pass* occurs, and a connection needs to be established with the *origin* server.

Here's an example:

```
-   BackendOpen    31 boot.default 172.22.0.2 8080 172.22.0.3 45378
-   BackendStart   172.22.0.2 8080
-   BackendClose   31 boot.default
```

The `BackendOpen` tag provides information about the *TCP* or *UDS* connection that is established with the backend. Here's the format of the tag:

```
%d %s %s %s %s %s
|  |  |  |  |  |
|  |  |  |  |  +- Local port
|  |  |  |  +---- Local address
|  |  |  +------- Remote port
|  |  +---------- Remote address
|  +------------- Backend display name
+---------------- Connection file descriptor
```

In our example file descriptor 31 is used for the backend, and the actual backend that was selected came from a *VCL configuration* named `boot`. Inside that *VCL configuration* a backend named `default` was used.

You'll agree that this is a very standard situation.

The connection with the *origin* was done via IP address `172.22.0.2` on port `8080`. The connection originated from IP address `172.22.0.3`, and the source port was `45378`. The `BackendStart` tag provides similar information and doesn't add that much value.

However, it is interesting to see that a `BackendClose` tag was found. This means the connection was not recycled for future requests to this backend.

If *keep-alive* was enabled, the origin would return a `Connection: Keep-Alive` header, and *Varnish* would reuse that connection. This would result in the `BackendReuse` tag to appear in *VSL*, as illustrated below:

```
-   BackendReuse   31 boot.default
```

Backend request tags

When a backend is opened, the goal is the send a backend request. The *backend request tags* are there to provide information about the backend request.

In most cases, the backend request will be identical to the client request. However, *VCL* does allow you to change request information, which might be reflected in the backend request.

These are the *backend request tags*:

- `BereqMethod`
- `BereqURL`
- `BereqProtocol`
- `BereqHeader`
- `BereqAcct`

This probably needs further explanation.

Backend response tags

Just like regular *response tags*, there are also *backend response tags*. These are displayed when the response wasn't served from cache but required a backend request.

These are the tags, and they look very similar to regular response tags:

- `BerespProtocol`
- `BerespStatus`
- `BerespReason`
- `BerespHeader`

Again, this needs further explanation.

Object tags

- `ObjProtocol`
- `ObjStatus`
- `ObjReason`
- `ObjHeader`

VCL tags

So far we've focused on input and output:

- What request information is received by *Varnish*?
- What backend request information is sent to the *origin*?
- What backend response does the *origin* provide?
- What response are we sending back to the *client*?

Although this is very useful, we also need to focus on what happens within *Varnish*.

Luckily, there are some *VCL*-related tags that do just that. Here's a list of those tags:

- `VCL_Error`: returns the error message in case of a *VCL* execution failure
- `VCL_Log`: custom log messages that were logged via `std.log()` in *VCL*
- `VCL_acl`: evaluation of *ACLs* in *VCL*
- `VCL_call`: the name of the *VCL* state that is currently being executed
- `VCL_return`: the *return statement* that was used to transition to the next state
- `VCL_use`: the name of the *VCL* configuration that is being used

Here's an example containing some *VCL tags*:

```
*   << Request  >> 2
-   VCL_call       RECV
-   VCL_return     hash
-   VCL_call       HASH
-   VCL_return     lookup
-   VCL_call       MISS
-   VCL_return     fetch
-   Link           bereq 3 fetch
-   VCL_call       DELIVER
-   VCL_return     deliver
**  << BeReq    >> 3
--  VCL_use        boot
--   VCL_call       BACKEND_FETCH
--   VCL_return     fetch
--   VCL_call       BACKEND_RESPONSE
--   VCL_return     deliver
```

If you remember the *VCL flow* from earlier in the book, you'll notice that this log extract represents a *cache miss*.

The VCL_call tags go from RECV, to HASH, to MISS, to BACKEND_FETCH, to BACKEND_RESPONSE, to DELIVER. The VCL_return tags are responsible for this sequence of *VCL subroutines* being called.

As mentioned before, a *cache miss* is not necessarily a bad thing. It's just a *cache hit* that hasn't happened yet. And as you can see in the log extract below, the next request will result in a lot less output:

```
*   << Request  >> 5
-   VCL_call       RECV
-   VCL_return     hash
-   VCL_call       HASH
-   VCL_return     lookup
-   VCL_call       HIT
-   VCL_return     deliver
-   VCL_call       DELIVER
-   VCL_return     deliver
```

This is because this sequence of events represents a *cache hit*. No need to open up a << BeReq >> transaction and fetch data from the backend because the object can be served from cache.

The VCL_acl tag can contain the following information when an *ACL* wasn't successfully matched:

```
-   VCL_acl        NO_MATCH purge
```

In this case an *ACL* named purge couldn't match the IP address of the client.

The next example reflects a successful *ACL* match on the purge *ACL* for a client whose client.ip value could be matched to localhost:

```
-   VCL_acl        MATCH purge "localhost"
```

If you can, use `std.log()` in your *VCL* to log custom messages. Here's some *VCL* that was used to log *ACL* mismatches with some extra information:

```
std.log(client.ip + " didn't match the 'purge' ACL");
```

This log line appears in *VSL* via a `VCL_Log` tag.

```
-   VCL_Log        172.28.0.1 didn't match the 'purge' ACL.
```

And if you have multiple *VCL* configurations registered in *Varnish*, the `VCL_use` tag will remind you which ones were being executed.

Here's the standard value for that tag:

```
--- VCL_use        boot
```

If you didn't register any extra *VCL files*, `boot` would be the one that was used when `varnishd` was booted.

When an error occurs within your *VCL* configuration, the `VCL_Error` tag is displayed, containing the error that occurred.

Here's an example where too many levels of *ESI* were used:

```
-6- VCL_Error      ESI depth limit reach (param max_esi_depth = 5)
```

The timestamp tag

The `timestamp` tag is very important, as it shows how long individual aspects of the HTTP request took.

Here's the format of this tag:

```
%s: %f %f %f
|   |  |  |
|   |  |  +- Time since last timestamp
|   |  +---- Time since start of work unit
|   +------- Absolute time of event
+----------- Event label
```

Here's an example of the timing for a typical request that resulted in a *cache miss*:

```
*   << Request  >> 11
-   Timestamp      Start: 1606398588.811189 0.000000 0.000000
-   Timestamp      Req: 1606398588.811189 0.000000 0.000000
-   Timestamp      Fetch: 1606398588.818399 0.007210 0.007210
-   Timestamp      Process: 1606398588.818432 0.007243 0.000032
-   Timestamp      Resp: 1606398588.818609 0.007421 0.000178
**  << BeReq    >> 12
--  Timestamp      Start: 1606398588.811381 0.000000 0.000000
--  Timestamp      Bereq: 1606398588.814423 0.003042 0.003042
--  Timestamp      Beresp: 1606398588.817982 0.006601 0.003559
--  Timestamp      BerespBody: 1606398588.818372 0.006991 0.000390
```

Different aspects of the execution flow are timed:

- `Start`: when did the request start?
- `Req`: how long after the start did we receive the request?
- `Fetch`: in the case of a *miss*, how long did the *fetch* take?
- `Process`: how long did the processing of the *backend response* take?
- `Resp`: how long until we can return the response to the client that requested it?

The same applies to the `<< BeReq >>` transaction:

- `Start`: when did the backend request start?
- `Bereq`: how long after the start did we send out the *backend request*?
- `Beresp`: how long did it take for the *backend response* to arrive?
- `BerespBody`: how long until we processed the response body from the *backend response*?

The *Unix timestamps* in the second column gives you the exact time when each measurement took place with microsecond precision. However, we care more about the duration than the absolute time of execution.

The third column is the total time of execution since the start, and the fourth is the duration of this specific unit of work.

In our example it took the backend *72 milliseconds* to respond, and it took *Varnish* a little over *74 milliseconds* to return the response. But the individual task of returning the response to the client only took *178 microseconds*.

For a cached object, the following logs could be generated:

```
*    << Request  >> 14
-    Timestamp      Start: 1606399284.176260 0.000000 0.000000
-    Timestamp      Req: 1606399284.176260 0.000000 0.000000
-    Timestamp      Process: 1606399284.176393 0.000133 0.000133
-    Timestamp      Resp: 1606399284.176648 0.000388 0.000255
```

Because this was a *cache hit*, there is no `Fetch` timestamp tag. There's also no `<< BeReq >>` transaction. The total execution time of the request is a mere *388 microseconds*.

The TTL tag

Earlier in this book, we explained extensively how the *TTL* of an object is calculated:

- The *TTL* of a cached object can be set via *HTTP headers*.
- The *TTL* of a cached object can be overridden in *VCL*.
- The object might not be cacheable at all, which results in a *hit-for-pass* or *hit-for-miss* object.

All this information can be found in the *TTL* tag. Here's its format:

```
%s %d %d %d %d [ %d %d %u %u ] %s
|  |  |  |  |    |  |  |  |    |
|  |  |  |  |    |  |  |  |    +- "cacheable" or "uncacheable"
|  |  |  |  |    |  |  |  +------ Max-Age from Cache-Control header
|  |  |  |  |    |  |  +--------- Expires header
|  |  |  |  |    |  +------------ Date header
|  |  |  |  |    +--------------- Age (incl Age: header value)
|  |  |  |  +-------------------- Reference time for TTL
|  |  |  +----------------------- Keep
|  |  +-------------------------- Grace
|  +----------------------------- TTL
+-------------------------------- "RFC", "VCL" or "HFP"
```

Let's throw in a couple of examples to show the different values.

Here's the first one:

```
--  TTL            RFC 120 10 0 1606398419 1606398419 1606398419 0 0
cacheable
```

This object is *cacheable* and is stored in the cache for *120 seconds*. Because it is cacheable, it is definitely not a `HFP` object. But because no `Expires` or `Cache-Control` header was set, *Varnish* will fall back on its `default_ttl` value, which is set at two minutes.

Also interesting to note is that *ten seconds of grace* was added but no extra *keep time*.

Here's another cacheable response:

```
--  TTL            RFC 25 10 0 1606400425 1606400425 1606400425 0 25
cacheable
```

The *TTL* for this example was set to *25 seconds*, and that was because the `max-age` value of the `Cache-Control` header was set to *25*.

Here's an example of a *hit-for-miss* object being created after a `Cache-Control: private, no-cache, no-store` header was received:

```
--  TTL            VCL 120 10 0 1606400537 uncacheable
```

For the next *two minutes*, all requests for this object will bypass the waiting list and will directly hit the backend. If in the meantime a cacheable response is returned, a regular object is inserted.

The more aggressive version of this is *hit-for-pass*. The following example is uncacheable for the same reasons but because `return(pass(10s));` was added as an explicit return statement, the *hit-for-miss* is turned into a *hit-for-pass*:

```
--  TTL            HFP 10 0 0 1606402666 uncacheable
```

The *TTL* value is *ten seconds* because of the fact that this duration was explicitly used in `return(pass(10s));`.

In the final *TTL* example, we'll juice up *grace* and *keep* a bit:

```
--  TTL            RFC 500 10 0 1606403184 1606403184 1606403183 0
500 cacheable
```

The `Cache-Control` header had a `max-age` value of *500 seconds*; we set the *grace* to an hour, and the *keep* to a day.

The fact that we set the *TTL* via an *HTTP header* turned this into an `RFC` log item.

7.8.4 Output filtering

Now that we've familiarized ourselves with the various *VSL tags*, it's time to put these tags to use.

If you run `varnishlog` on a production server, you'll be overwhelmed by the amount of output coming your way. By filtering out specific tags, the information is easier to process.

Tag inclusion

The `-i` parameter in `varnishlog` will only include the tags that were mentioned.

Here's a standard example, where we want to know the *URL* of a request and its flow through the *finite state machine*:

```
varnishlog -i ReqUrl,VCL_call,VCL_return -g session
```

This example will only include the `ReqUrl`, `VCL_call`, and `VCL_return` tags for transactions that were grouped by session:

```
*   << Session  >> 1
**  << Request  >> 2
--  ReqURL         /
--  VCL_call       RECV
--  VCL_return     hash
--  VCL_call       HASH
--  VCL_return     lookup
--  VCL_call       MISS
--  VCL_return     fetch
--  VCL_call       DELIVER
--  VCL_return     deliver
*** << BeReq    >> 3
--- VCL_call       BACKEND_FETCH
--- VCL_return     fetch
--- VCL_call       BACKEND_RESPONSE
--- VCL_return     deliver
```

As you can see, the homepage was consulted but was not served from cache. A backend fetch was required.

A very common example, and thanks to output filtering, the logs are easier to interpret.

Wildcards are also supported. Our previous example can even be rewritten as follows:

```
varnishlog -i "ReqUrl,VCL_*" -g session
```

Tag exclusion

You can also exclude tags from the output. This is done by using the `-x` parameter.

Here's an example where we include all tags that start with `Req`, but we want to exclude the `ReqHeader` and `ReqUnset` tags:

```
varnishlog -i "Req*" -x ReqHeader,ReqUnset
```

And here's the output:

```
-   ReqStart       172.28.0.1 52280 http
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqAcct        140 0 140 294 608 902
```

Tag inclusion by regular expression

Basic tag inclusion and exclusion is already a step in the right direction. But some tags have many occurrences in a single transaction.

Take for example a request where you only care about the `Accept-Language` header. Including `ReqHeader` can create a lot of noise.

The solution is to filter out tags by value. The `-I` uppercase parameter does just that.

The following example retrieves the URL of the request and the `Accept-Language` header:

```
varnishlog -g request -i ReqUrl -I ReqHeader:Accept-Language
```

Here's the output for the next request:

```
-   ReqURL         /
-   ReqHeader      Accept-Language: en-US,en;q=0.9,nl;q=0.8,nl-
NL;q=0.7
```

If it weren't for the -I parameter, you'd get a lot more irrelevant output, even if you use -i.

Tag exclusion by regular expression

Whereas -I includes tags by matching the value, the same thing can be done for exclusion. The -X uppercase parameter can be used to exclude tags based on a regular expression.

Here's an example where we include the ReqUrl and RespHeader tags, but we exclude all response headers that start with an *X*, both uppercase and lowercase:

```
varnishlog -g request -i ReqUrl -i RespHeader -X "RespHeader:(X|x)-"
```

Here's the output for the next request:

```
*   << Request  >> 5
-   ReqURL         /
-   RespHeader     Content-Type: application/json; charset=utf-8
-   RespHeader     Content-Length: 539
-   RespHeader     ETag: W/"21b-faj3J9Bg0wmX965fRcvtQFqPZr4"
-   RespHeader     Date: Fri, 27 Nov 2020 09:19:48 GMT
-   RespHeader     Age: 99
-   RespHeader     Accept-Ranges: bytes
-   RespHeader     Connection: keep-alive
```

The X-Varnish response header is removed from this transaction, possibly along with some other headers that were sent by the *origin* that started with an *X*.

Filtering by request type

Unless the -c or -b flags were added to varnishlog, both << Request >> and << BeReq >> transactions are included in the output.

However, you can filter out entire transactions:

- If you only care about the *client-side request*, you can use the -c parameter.
- If you only care about the *backend request*, you can use the -b parameter.

Using both `-c` and `-b` will include both types of requests, but that has the same effect as not mentioning them at all.

The following example will list both the *URL* that was provided by the client, and the *URL* that was sent to the *origin*. However, the `-c` flag will prevent the `<< BeReq >>` transaction from being included in the output:

```
varnishlog -g request -c -i ReqUrl -i BereqUrl
```

Here's the output:

```
*   << Request  >> 12
-   ReqURL         /
```

The all-in-one example

Let's end the output filtering subsection with an *all-in-one* example that uses all filtering techniques.

Here's the `varnishlog` command:

```
varnishlog -c -g request -i Req* -i Resp* \
    -I Timestamp:Resp -x ReqAcct -x RespUnset \
    -X "RespHeader:(x|X)-(url|host)"
```

And before we display the log information, we need to add some context to the story.

The *VCL* for this example adds two custom response headers: `x-url` and `x-host`. Before delivering the content to the client, these headers are stripped off again.

If you remember *asynchronous bans* from *chapter 6*, you'll know that *request context* needs to be injected in the *response object*; otherwise the *ban lurker* cannot ban objects based on the *URL* and *hostname*.

As a reminder, here's the *VCL* that adds and removes these headers:

```
sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
    set beresp.http.x-host = bereq.http.host;
}

sub vcl_deliver {
    unset resp.http.x-url;
    unset resp.http.x-host;
}
```

These actions are also reported in *VSL* through `RespHeader` and `RespUnset` tags. The `RespUnset` tag reports response tags being removed.

But we don't want our *VSL* output to be polluted with this kind of information, hence the `-x RespUnset` and `-X "RespHeader:(x|X)-(url|host)"` parameters.

With that in mind, here's the output:

```
*   << Request  >> 1
-   ReqStart       172.18.0.1 38076 http
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   ReqHeader      User-Agent: curl/7.64.1
-   ReqHeader      Accept: */*
-   ReqHeader      X-Forwarded-For: 172.18.0.1
-   RespProtocol   HTTP/1.1
-   RespStatus     200
-   RespReason     OK
-   RespHeader     Content-Type: application/json; charset=utf-8
-   RespHeader     Content-Length: 555
-   RespHeader     ETag: W/"22b-nCrko0g3BQi5EC4Z9AcxPigbGms"
-   RespHeader     Date: Fri, 27 Nov 2020 09:40:46 GMT
-   RespHeader     X-Varnish: 65543
-   RespHeader     Age: 100
-   RespHeader     Via: 1.1 varnish (Varnish/6.0)
-   RespHeader     Accept-Ranges: bytes
-   RespHeader     Connection: keep-alive
-   Timestamp      Resp: 1606470046.159483 0.003742 0.000081
```

The log output gives us request information, response information, and the total duration of execution.

7.8.5 VSL queries

So far we've been filtering output and have only been displaying the tags we care about. This definitely decreases the amount of log lines in the output.

But there's still the potential for noise. Although we've been reducing the size of the transactions, we're still displaying all transactions.

In this section we're going to apply *VSL queries* to include only transactions that match specific criteria.

The `-q` parameter will allow us to specify a query. The syntax for *VSL queries* goes as follows:

```
<record selection criteria> <operator> <operand>
```

Yes, this is quite vague. An example will make this make more sense. Here it is:

```
varnishlog -i VCL_call -i VCL_return -q "ReqUrl eq ‘/’"
```

This `varnishlog` command will display the *VCL flow* by including `VCL_call` and `VCL_return` tags, but only for the homepage, that is, only when the `ReqUrl` tags is equal to `/`.

If we refer back to the *VSL query syntax*, we can break this down as follows:

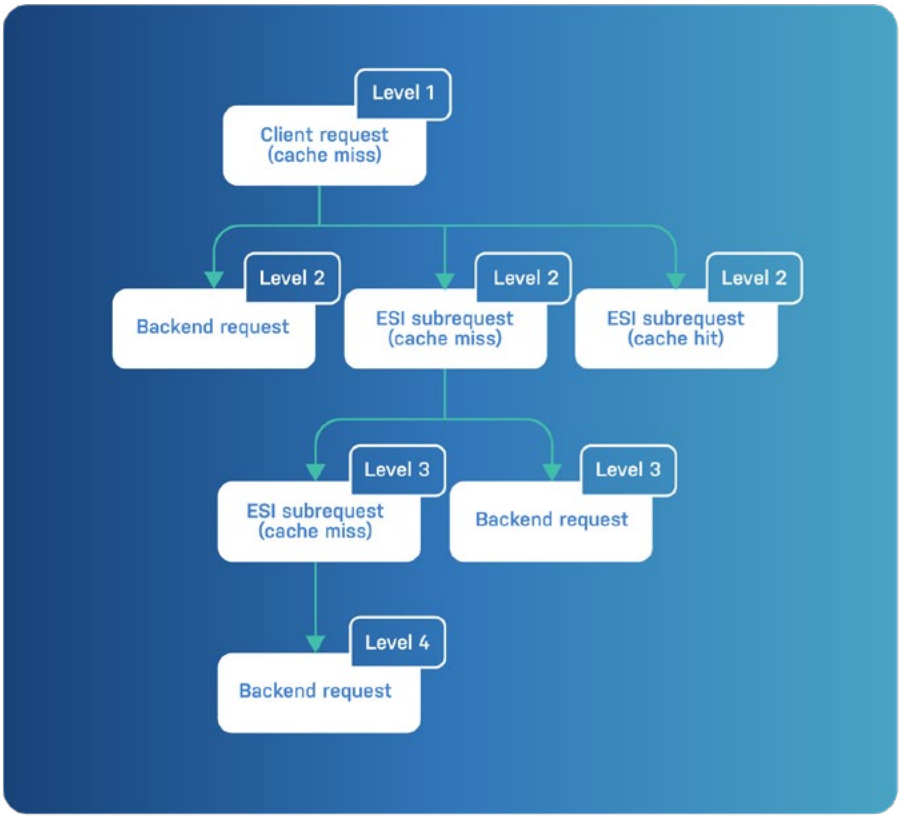
- `ReqUrl` is part of the record selection criteria.
- `eq` is the operator.
- `‘/’` is the operand.

Record selection criteria

The *record selection criteria* can be a lot more elaborate than what we just showed you. Here's the syntax for these criteria:

```
{level}taglist:record-prefix[field]
```

The `{level}` syntax refers to the transaction hierarchy in *VSL*. Here's a diagram that clarifies these levels:


Level 1
Client request (cache miss)
Level 2
Backend request
Level 2
ESI subrequest (cache miss)
Level 2
ESI subrequest (cache hit)
Level 3
ESI subrequest (cache miss)
Level 3
Backend request
Level 4
Backend request

VSL transaction levels

These levels only apply when *request grouping* takes place. And the *level 1* transaction is the *client request*.

When a cache miss takes place, a *level 2* transaction appears in the logs. This contains a *backend request*, but at the same time can also contain a set of *ESI subrequests*.

In the diagram, we've made a distinction between *ESI subrequests* that resulted in a *cache hit* or a *cache miss*.

For *ESI subrequests* that cause a *cache miss*, a *level 3* backend request is required. Maybe that *ESI subrequest* triggers another *ESI subrequest*.

If that *level 3 ESI subrequest* also results in a *cache miss*, a *level 4* backend request is triggered.

This can go on and on until you hit the `max_esi_depth` limit. The default value for the `max_esi_depth` runtime parameter is currently *five*.

The `taglist` directive refers to one tag or a *glob pattern* that matches multiple tags.

If a transaction has multiple occurrences of a tag, the `record-prefix` can be used to single out log lines that match the prefix.

And if you want to match specific values from individual fields in a log line, the `[field]` syntax can be used.

This is a lot of information to digest. Let's throw in an example that uses the full syntax:

```
varnishlog -c -i ReqUrl -I Timestamp:Resp -g request -q "{2+}
Time*:Resp[2] > 2.0"
```

This example will only display transactions that have subtransactions. The number of levels doesn't matter as long as it is more than two.

For these transactions, only the client-side transaction is shown, and only the *request URL* and *timestamp* are displayed for responses that took longer than two seconds to generate.

Let's break down the query:

- `{2+}`: the query applies to transactions at level 2 or greater
- `Time*`: a glob pattern that matches all tags that start with `Time`
- `:Resp`: refers to prefixes of log lines that match `Resp` for these tags
- `[2]`: looks at the second field of a matched log line
- `> 2.0`: ensures that value is greater than two

And let's have a look at parameters as well:

- `-c`: only displays *client-side* transactions
- `-i ReqUrl`: displays the URL of the request
- `-I Timestamp:Resp`: displays `Timestamp` tags that have a `Resp` prefix
- `-g request`: groups the transactions by request
- `-q`: performs a *VSL query*, and only displays transactions that match the query

This could be the output of the `varnishlog` command:

```
*   << Request  >> 13
-   ReqURL         /
-   Timestamp      Resp: 1606733707.261102 3.012484 3.007354
**  << Request  >> 15
--  ReqURL         /esi
--  ReqURL         /esi
--  Timestamp      Resp: 1606733707.260886 3.006929 0.000161
```

As we can see, the `/` page takes more than three seconds to load and that is because the `/esi` subrequest took so long to load.

Operators

We've already seen some *VSL query* examples that feature both string comparison and numerical comparison.

For the sake of completeness, here's a list of the *operators* that are supported by the *VSL query syntax*:

- `==`: the operand numerically equals the record value
- `!=`: the operand doesn't numerically equal the record value
- `<`: the operand is greater than the record value
- `<=`: the operand is greater than or equal to the record value
- `>`: the operand is less than the record value
- `>=`: the operand is less than or equal to the record value
- `eq`: the operand equals the record string value
- `ne`: the operand doesn't equal the record string value
- `~`: the record value matches the regular expression pattern
- `!~`: the record value doesn't match the regular expression pattern

Operands

We've already covered the *record selection criteria*, and we listed the *operators*. We just need to talk about *operands*, which is very straightforward.

There are four types of operands:

- Integers
- Floating point numbers
- Strings
- Regular expressions

Let's have an example for each operand type. Here's the one for the *integer* type:

```
varnishlog -g request -q "BerespStatus >= 500"
```

This example will only show transactions whose status code is greater than or equal to 500. This implies retrieving server errors.

Here's the *float* example:

```
varnishlog -g request -q "Timestamp:Resp[2] > 2.0"
```

This example command looks familiar and was already featured. Just remember that the timestamps have microsecond precision and are expressed as *floating point numbers*.

The string example is also quite simple:

```
varnishlog -g request -q "ReqUrl eq '/'"
```

This example will only show transactions for the homepage.

And here's the regular expression example:

```
varnishlog -g request -q "ReqUrl ~ '^/contact'"
```

This example will match all transactions that start with `/contact`. This also includes requests for a URL like `/contact-page`.

You may wonder why *booleans* aren't included in the list of types. This is because transactions don't have boolean values. However, boolean comparisons are supported.

Here's an example of a boolean comparison:

```
varnishlog -g request -q "ReqHeader:Accept-Language"
```

This query will include all transactions that have an `Accept-Language` request header. The exact opposite is also supported by adding the `not` keyword:

```
varnishlog -g request -q "not ReqHeader:Accept-Language"
```

This example will include all transactions that do not have an `Accept-Language` request header.

Chaining queries

All the *VSL query* examples we've featured so far used a single comparison.

Multiple comparisons are supported, and queries can be chained using the `and` boolean function, or the `or` boolean function.

Here's an example that combines two comparisons:

```
varnishlog -c -i ReqUrl -I RespHeader:Content-Type \
    -i reqacct -g request \
    -q "RespHeader:Content-Type ~ '^image/' and ReqAcct[5] >=
2000000"
```

This example will display the URL, `Content-Type` header, and *request handling byte counts* for all transactions where the `Content-Type` response header matches the `^image/` pattern, and the number of bytes returned by the body is greater than *2 MB*.

Long story short: we're displaying transactions for images larger than 2 MB.

Here's the potential output:

```
*   << Request  >> 65562
-   ReqURL         /image.jpg
-   RespHeader     Content-Type: image/jpeg
-   ReqAcct        82 0 82 311 2832889 2833200
```

As you can see, the `Content-Type` header is `image/jpeg`, which matches the regular expression. And also the fifth field of the `ReqAcct` tag is `2832889`, which is more than *2 MB*. The combination of these two comparisons results in the output.

Here's an example where the `or` boolean function is used to chain comparisons:

```
varnishlog -c -g request -i ReqUrl \
    -I VCL_call:PASS -I VCL_call:MISS \
    -q "VCL_call eq 'MISS' or VCL_call eq 'PASS'"
```

This example will show the URL and *MISS/PASS* status for all requests that are cache misses, or where the cache is bypassed.

Here's the potential output for such a query:

```
*   << Request  >> 10
-   ReqURL         /
-   VCL_call       MISS
*   << Request  >> 525
-   ReqURL         /account
-   VCL_call       PASS
```

And finally, parentheses can be used when `and` and `or` are combined. Here's an example where we use parentheses for this:

```
varnishlog -c -g request -i ReqUrl \
    -q "TTL[6] eq 'uncacheable' and (BerespHeader:Set-Cookie or Bere-
spHeader:Cache-Control ~ '(private|no-cache|no-store)')"
```

This example will display the URL of the corresponding request for uncacheable responses that were caused by the presence of a `Set-Cookie` response header, or the fact that the `Cache-Control` response header contained non-cacheable directives.

7.8.6 Other VSL options

Up until this point we primarily focused on filtering and querying options. Tools like `varnishlog`, `varnishtop`, and `varnishncsa` also have some other useful options, which we'll discuss now.

> We'll primarily apply them to `varnishlog`, but `varnishtop` and `varnishncsa` will also covered separately.

Processing the entire buffer

When *VSL programs* are run, input is collected from the moment the program is started. However, there is already a lot more information in the *VSL circular buffer*.

By adding the `-d` option to `varnishlog`, `varnishtop`, or `varnishncsa`, the output starts at the head of the log and exits.

Here are the corresponding examples:

```
varnishlog -i ReqUrl -d
varnishtop -i ReqUrl -d
varnishcsa -d
```

The `-d` option is useful for getting the full picture. The size of the VSL space is controlled by the `-l` option for `varnishd`, and as stated previously the default size is *80 MB*. The bigger you set this, the more information is held in this buffer.

Here's an example where you get the *URL* and *starting timestamp* for all transactions in the *VSL buffer* that started at `1/12/2020 11:00:00 UTC` or later:

```
varnishlog -g request -i requrl -d \
    -q "Timestamp:Start[1] >= $(date -d '1/12/2020 11:00:00' +%s.0)"
```

The `$(date -d '1/12/2020 11:00:00' +%s.0)` subcommand was used to convert the date and time into the *Unix timestamp* format that is used by the `Timestamp` tag.

Dumping the contents of the buffer is a lot more interesting when you aggregate the data using `varnishtop`:

```
varnishtop -i ReqUrl -d -1
```

In this example the top request URLs are computed. `-d` will dump the buffer and use this as input. The `-1` ensures the computation only happens once, and the output is sent to *standard output*, instead of constantly being refreshed.

This is the potential output:

```
6582.00 ReqURL /
3920.00 ReqURL /contact
2640.00 ReqURL /products
```

The homepage is by far the most popular page with an average request rate of *6582 requests per second*.

Here's an extract of the output if `varnishncsa -d` is used to process the *VSL buffer*:

```
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
products HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
contact HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
products HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
contact HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
contact HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
contact HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
contact HTTP/1.1" 200 6 "-" "curl/7.64.1"
192.168.0.1 - - [01/Dec/2020:12:11:31 +0000] "GET http://localhost/
products HTTP/1.1" 200 6 "-" "curl/7.64.1"
```

This output is different from `varnishlog` because it formats it in *Apache/NCSA* format. This format is suitable for access logs.

Rate limiting

VSL is extremely verbose. Running a tool like `varnishlog` on a production server will become somewhat overwhelming.

Tag filtering and using *VSL queries* to reduce the output will only get you so far. On busy systems where a lot of transactions meet the querying criteria, there will still be tons of output.

If you don't need every single log line to get the required insight, you can opt to use rate limiting.

The `-R` uppercase parameter will limit the number of transactions that are returned based on a ratio.

Here's an example where only ten transactions per minute are displayed:

```
varnishlog -g request -R 10/1m
```

The format of `-R` is `<number-of-requests>/<duration>`. The *duration* is expressed as a *VCL duration type*. This includes the numerical value and the time unit suffix.

The same limits can be imposed on a `varnishncsa` command, as illustrated below:

```
varnishncsa -R 10/1m
```

Rate limiting is not available for `varnishtop`, which makes sense. The `varnishtop` program aggregates the output and is not that noisy. The results would be heavily skewed if rate limiting were to be applied.

Storing and replaying logs

The output from `varnishlog` can be stored in a file. This can either be done in a *human-readable format* for analysis later on, but we can also store it in a *binary format*.

Logs in binary format can be replayed in `varnishlog` where filters and queries can be applied after the fact.

The easiest way to store the log in a file is by running the following command:

```
varnishlog -w vsl.log
```

As long as the command is running, *VSL logs* will be stored in `vsl.log`. When this command is run again, `vsl.log` will be overwritten.

The logs are persisted in a binary format, which makes replaying them quite easy.

Replaying those logs is done using the following command:

```
varnishlog -r vsl.log
```

And as mentioned before, queries and filters can be applied. Here's an example:

```
varnishlog -i ReqUrl -i ReqAcct \
    -q "RespHeader:Content-Type ~ ‘^(image|video)/’" \
    -r vsl.log
```

This example will read `vsl.log` and will find all transactions where the `Content-Type` header of the response matches images and video files. For those transactions the *URL* and the *byte count sizes* are returned.

When you want to ensure that your log file is not overwritten every time the `varnishlog -w vsl.log` is called, you can use the `-a` parameter. This parameter ensures that data is appended to the log file every time the command is called.

Remember that anything filtered out when writing log files will be missing during replay. We advise you not to use the `-i` and `-I` options in this situation, and just let `varnishlog` write the complete transaction.

VSL queries are allowed, but the full transaction is expected to be written to file. Please also watch out what kind of *transaction grouping* you use. If you use `-g request`, please understand that *session grouping* is not supported when you replay the logs because you are effectively filtering sessions out.

The `-g raw` grouping is the fastest one and can be useful to increase write throughput, but it also increases the chance of collecting partial transactions that wouldn't be useful during replay. Since raw grouping will capture both transactional and non-transactional logs, you might want to clarify this with a query. For example to only collect non-transactional logs:

```
varnishlog -g raw -q ‘vxid == 0’
```

If you don't care about log replaying and applying filters and queries after the fact, you can also store `varnishlog` output in *ASCII format*. Just add the `-A` uppercase parameter to make this happen. Remember that you will lose the ability to perform structured querying or filtering.

Let's revisit the example where we're inspecting the URL and *byte count sizes* for images and video files, and append that information in a *human-readable format* to `vsl.log`:

```
varnishlog -i ReqUrl -i ReqAcct \
    -q "RespHeader:Content-Type ~ ‘^(image|video)/’" \
    -w vsl.log -a -A
```

7.8.7 varnishncsa

We spent a lot of time talking about `varnishlog`. It's an important tool, maybe even the most important logging tool in our toolkit.

But let's not forget that `varnishncsa` is also a very powerful tool. Perhaps not as verbose as `varnishlog`, but the advantage of `varnishncsa` is usually a *single-line*, concise output*.

This output is formatted in the *Apache/NCSA combined format*. This is what the format looks like by default:

```
%h %l %u %t "%r" %s %b "%{Referer}i" "%{User-agent}i"
```

- `%h`: the hostname or IP address of the remote host
- `%l`: remote log name
- `%u`: remote authenticated user
- `%t`: time when the request was received
- `%r`: first line of the HTTP request
- `%s`: HTTP status code of the response
- `%b`: size of the response body in bytes
- `%{Referer}i`: the `Referer` request header
- `%{User-Agent}i`: the `User-Agent` request header

Here's some potential output in this format:

```
192.168.0.1 - - [01/Dec/2020:13:45:23 +0000] "GET http://localhost/
HTTP/1.1" 200 6 "-" "curl"
```

By default `varnishncsa` will only output *client transactions*. That is because `varnishncsa` is there to replace your regular web server logs. Because of caching and the fact that requests are offloaded from the origin server, the logs you usually consult will remain mostly empty. The higher the hit rate of your cache, the lower the number of log lines.

Standard *NCSA-style* log files are stored on disk. By default `varnishncsa` just consults the *VSL circular memory buffer*, so there is no disk access.

It is quite common to store `varnishncsa` output on disk. Here's an example:

```
varnishncsa -a -w /var/log/varnish/access.log
```

If your *pre-Varnish setup* used log analysis or log centralization tools, `varnishncsa` log files are an excellent surrogate.

Logging modes

As mentioned, the standard logging mode will only output *client transactions*. It is possible to display *backend transactions* by adding the `-b` parameter to `varnishncsa`.

If you want to combine *client and backend transactions* in your output, you can add both the `-c` and `-b` parameters. However, the output may be confusing.

You can add the `%{Varnish:side}x` formatter to your output. This formatter will return `c` for client requests and `b` for backend requests.

This is the corresponding command to create that sort of output:

```
varnishncsa -c -b -F '%h %l %u %t "%r" %s %b "%{Referer}i" "%{Us-
er-agent}i" "%{Varnish:side}x"'
```

And here's some example output for this command:

```
172.18.0.3 - - [02/Dec/2020:10:18:38 +0000] "GET http://localhost/
HTTP/1.1" 200 6 "-" "curl" "b"
127.0.0.1 - - [02/Dec/2020:10:18:38 +0000] "GET http://localhost/
HTTP/1.1" 200 6 "-" "curl" "c"
```

As you can see the `%h` formatter that outputs the remote host is different for each of the lines. Their meaning is also a bit different, depending on the mode:

- In *backend mode*, the `%h` formatter refers to the IP address of the backend server.
- In *client mode*, the `%h` formatter refers to the IP address of the client.
- There's also a difference in meaning for the `%b` formatter:
- In *backend mode*, the `%b` formatter refers to the number of bytes received from the backend for the response body.
- In *client mode*, the `%b` formatter refers to the total byte size of the response for the response body.

If the response body is modified in *VCL*, these counters can have a different value.

Modifying the log format

The standard format for `varnishncsa` is the *NCSA combined log format*. We've described the format earlier. It is a conventional format that most log analyzers support.

However, you're not obligated to respect that format. For the sake of completeness, here is list of all supported formatters:

Formatter	Meaning
`%b`	Size of the response body in bytes
`%D`	Time taken to serve the request in microseconds
`%H`	The request protocol
`%h`	The hostname or IP address of the remote host
`%I`	Total bytes received
`%{X}i`	Contents of request header `X`
`%l`	Remote log name
`%m`	Request method
`%{X}o`	Contents of response header `X`
`%O`	In client mode, total bytes sent to client. In backend mode, total bytes received from the backend
`%q`	The query string
`%r`	First line of the HTTP request composed using other formatters
`%s`	HTTP status code of the response
`%t`	Time when the request was received
`%{X}t`	Time when request was received in the `strftime` time specification format
`%T`	Time taken to serve the request
`%U`	Request URL without query string
`%u`	Remote authenticated user
`%{X}x`	Extended Varnish & VCL variables

The `-F` parameter allows you to specify a format string. Here's a simple example:

```
varnishncsa -F '%U %T'
```

This format will return the URL of the request and the time in seconds it took to serve the request.

You can also save your log format in a file. If we revisit the example where `%{Varnish:side}x` was added to the format, we could store the following content in `/etc/varnish/varnishncsa_combined_and_side_formatter`:

```
%h %l %u %t "%r" %s %b "%{Referer}i" "%{User-agent}i"
"%{Varnish:side}x"
```

Loading this format can be done via the `-f` parameter, as illustrated below:

```
varnishncsa -c -b -f /etc/varnish/varnishncsa_combined_and_side_for-
matter
```

This way you can store multiple formats in separate files and easily call them when required.

There are also some special formatters that allow you to inject headers and other special values.

Let's have a look at headers first. The `%{X}i` formatter replaces the `X` with a valid *request header* and returns its values. The same applies to `%{X}o`, where the `X` is replaced with a valid *response header*.

Here's an example where we care about the *URL*, the `Accept-Language` request header, and the `Content-Type` response header:

```
varnishncsa -F ‘%U "%{Accept-Language}i" "%{Content-Type}o"’
```

This is the output that is generated:

```
/ "en-US,en;q=0.9,nl;q=0.8,nl-NL;q=0.7" "text/html;
charset=UTF-8"
```

And there is also another special one, which is `%(X)t`. This formatter allows you to return the request time in a custom time format. As stated in the table above, the `strftime` format is used. See the `strftime(3)` manual on your system by running `man 3 strftime` for more information about this format.

Here's an example where we display the full date, including the day of the week, and the hour of the day:

```
varnishncsa -F ‘%{%A %B %-d %Y %-I %p}t’
```

Here's some output we generated:

```
Wednesday December 2 2020 11 AM
```

Let's break down the *date format*:

- `%A`: the full name of the day of the week
- `%B`: the full month name
- `%-d`: the day of the month as a decimal number without the zero-padding
- `%Y`: the year as a decimal number including the century
- `%-I`: the hour as a decimal number using a 12-hour clock without the zero-padding
- `%p`: *AM* or *PM*

Extended variables

There is one type of formatter that deserves its own subsection. The *extended variables* formatter, which uses the `%{X}x` notation, allows us to use specific variables and access *VSL* information. The sky is the limit, basically.

Let's look at the extended variables first before talking about *VSL*.

Here's the list of variables that can be accessed:

- `Varnish:time_firstbyte`: time until the first byte is returned
- `Varnish:hitmiss`: *hit/miss* marker
- `Varnish:handling`: returns either `hit`, `miss`, `pass`, `pipe`, or `synth`. Indicates how the request was handled
- `Varnish:side`: returns `c` when in *client mode*, and `b` when in *backend mode*. We've already covered this one earlier
- `Varnish:vxid`: the *Varnish transaction ID*

Here's an example where `Varnish:handling` and `Varnish:side` are used. It gives you some insight as to what happened behind the scenes for this request.

Here's the command:

```
varnishncsa -F '%U %{Varnish:handling}x %{Varnish:side}x' -b -c
```

In the example, two requests for `/test` are being sent. This is the output:

```
/test - b
/test miss c
/test hit c
```

As you can see, the *backend transaction* doesn't decide on the handling. However, the *client transaction* does and qualifies it as a *cache miss*. For the second request, there is not a *backend transaction* because it was a *cache hit*.

The next request contained a *cookie*, based on the *built-in VCL*, which resulted in a *pass*:

```
/test - b
/test pass c
```

The final request was made using an unrecognized *request method*. The *built-in VCL* uses *piping* to send that request to the backend. As you may remember, *Varnish* no longer considers this an *HTTP request* and shuffles the bytes directly to the *origin*.

Here's the output:

```
/test - b
/test pipe c
```

And then there's the *VSL information* we can include. According to the specification, this is the format:

```
VSL:tag:record-prefix[field]
```

This should look familiar to you.

Here's a first example where we use `%b` to display the *body bytes* that were transmitted. But thanks to `%{VSL:ReqAcct[4]}x`, we can also display the *header bytes* that were transmitted:

```
varnishncsa -F "%b (body bytes) %{VSL:ReqAcct[4]}x (header bytes)"
```

This is the output:

```
3257 (body bytes) 276 (header bytes)
```

The following example will display the *TTL*, *grace*, and *keep* values for an object, along with its URL. Here's the command:

```
varnishncsa -b -F "%U TTL: %{VSL:TTL[2]}x - Grace: %{VSL:TTL[3]}x -
Keep: %{VSL:TTL[4]}x"
```

This is the output:

```
/ TTL: 100 - Grace: 10 - Keep: 0
```

VSL queries

VSL queries are not restricted to `varnishlog`. We can also use them in `varnishncsa` to narrow down the scope of what we want to log.

Once you start adding queries, this is no longer a surrogate for regular *access logs*. Chances are that you'll be running multiple `varnishncsa` services, depending on the things you want to log.

The `-q` parameter, which we covered extensively, is also used here. The following example will log backend responses that took more than a second to load:

```
varnishncsa -b -F '%{VSL:Timestamp:BerespBody[2]}x %{Host}i %U%q' -q
'Timestamp:BerespBody[2] > 1.0'
```

Here's the output:

```
2.007142 localhost /
```

This example only outputs *backend transactions* when the query matches. It would seem as though you have the same flexibility as with `varnishlog`. An extra benefit is single-line output. However, you're very strictly tied to the *mode* you're in.

Unless you combine `-c` and `-b`, there is no way to return *backend transaction* information and *client transaction* information at the same time. And if you do combine both modes, the output will be spread across two lines.

Other varnishncsa options

Just like `varnishlog`, `varnishncsa` also had the well-known `-w` and `-a` options to write and append logs to a file.

The `-d` option is also there to dump the entire *VSL buffer* and exit. And unsurprisingly the `-R` option is there to perform rate limiting.

Transaction grouping via `-g` is also supported but is limited to `-g request` and `-g vxid`.

You can even replay *binary VSL logs* from a file and present them in *NCSA combined format* using the `-r` option.

This is all very similar to `varnishlog` and needs no further explanation.

Log rotation

When you start logging incoming requests via `varnishncsa`, and these logs end up being stored in files, we have to be careful. On busy production systems, the size of these logs can become gigantic.

That's why a proper *log rotation strategy* is in order. Log rotation ensures that logs are automatically archived to avoid running out of disk space.

A number of logs are kept, based on certain criteria. Given the configured frequency, the current log file is moved and renamed, while an empty log file is used by `varnishncsa`. When the maximum number of log files is reached, the oldest one is removed. This is a cyclical process.

When you install *Varnish* via packages, a log-rotate configuration is available for `varnishncsa`.

This is what it looks like:

```
/var/log/varnish/varnishncsa.log {
  daily
  rotate 7
  compress
  delaycompress
  missingok
  postrotate
    systemctl -q is-active varnishncsa.service || exit 0
    systemctl reload varnishncsa.service
  endscript
}
```

This configuration is located in `/etc/logrotate.d/varnish` and is picked up automatically by the `logrotate` program.

This configuration will perform *daily* log rotation and rotates as soon as *seven log files* exist.

This is what the directory layout will look like:

```
varnishncsa.log
varnishncsa.log.1
varnishncsa.log.2.gz
varnishncsa.log.3.gz
varnishncsa.log.4.gz
varnishncsa.log.5.gz
varnishncsa.log.6.gz
varnishncsa.log.7.gz
```

Log files will be compressed via *gzip*, and compression is delayed until the next rotation. That's why `varnishncsa.log.1` is not compressed.

The `missingok` directive makes sure that `logrotate` doesn't complain when a log file is missing.

The commands between `postrotate` and `endscript` are tasks that can be performed after the log rotation. In our case, it is important to signal to `varnishncsa` that it can release the log file and reopen it. Otherwise `varnishncsa` will still attempt to write to the original file.

By sending a `SIGHUP` signal to the `varnishncsa` process, it will release and reopen. This requires prior knowledge about the *process ID* of the `varnishncsa` process.

If you were to manually run `varnishncsa` as a daemon, you could specify the `-P` parameter to write the *process ID* to a file.

Here's an example of running `varnishncsa` as a service but without a proper service manager supporting it:

```
varnishncsa -a -D -w /var/log/varnish/access.log -P /var/run/varnish-
ncsa.pid
```

The `-D` parameter will *daemonize* the process and `-P` will write the *PID file* to `/var/run/varnishncsa.pid`.

You can use the following `postrotate` command:

```
postrotate
  kill -HUP $(cat /var/run/varnishncsa.pid)
endscript
```

This basically sends a `SIGHUP` signal to the process identified by what's inside `/var/run/varnishncsa.pid` and forces `varnishncsa` to release and reopen the log file.

But on production systems, a *service manager* like `systemd` will be used to start `varnishncsa` for you and will have reload logic to handle the `SIGHUP`. The `systemctl reload varnishncsa.service` command handles this.

7.8.8 varnishtop

VSL is consumed by three programs. We already talked about `varnishlog` and `varnishncsa`. We still need to cover `varnishtop`.

The input that `varnishtop` consumes is the same: it reads from the *VSL circular memory buffer*. The output, on the other hand, is very different. `varnishtop` does not output log lines like the other tools. It presents a continuously updated list of the most commonly occurring log entries.

The *tags* that appear the most across transactions are displayed first.

Here's a raw output extract when running `varnishtop`:

```
list length 8517

 11125.37 VCL_return     deliver
  5562.96 Begin          sess 0 HTTP/1
  5562.78 RespStatus     200
  5562.78 RespReason     OK
  5562.78 VCL_call       HASH
  5562.78 RespProtocol   HTTP/1.1
  5562.78 VCL_call       DELIVER
  5562.78 VCL_return     lookup
  5562.78 RespHeader     Server: nginx/1.19.0
  5562.78 RespHeader     Connection: close
  5562.78 RespHeader     X-Powered-By: PHP/7.4.7
  5562.78 RespHeader     Accept-Ranges: bytes
  5562.78 ReqHeader      Accept-Encoding: gzip
  5562.78 RespHeader     Date: Wed, 02 Dec 2020 16:23:15 GMT
  5562.78 RespHeader     Via: 1.1 varnish (Varnish/6.0)
  5562.78 RespHeader     Cache-Control: public ,max-age=100
  5562.78 RespHeader     Content-Type: text/html; charset=UTF-8
  5562.73 ReqMethod      GET
```

```
    5562.73 VCL_call       RECV
    5562.73 VCL_return     hash
```

It comes as no surprise that the `VCL_return deliver` log line is the most popular. Given the traffic patterns of this example, it occurs about *11125.37* times per second.

We have to be honest here: this output is not very helpful without proper *tag filtering* and *VSL queries*.

Here's a more sensible `varnishtop` command:

```
varnishtop -i requrl
```

This command will list the most popular URLs. This could be the output:

```
list length 3

    31.67 ReqURL         /contact
    24.33 ReqURL         /products
     8.50 ReqURL         /
```

This output shows that the `/contact` page is the most popular one with an average of *31.67 requests per second*.

Other questions that may be answered with `varnishtop` are:

- Which URLs cause the most cache misses?
- Which URLs cause the most cache bypasses?
- What are the most popular uncacheable objects?
- What are the most consumed pages that take the origin more than two seconds to generate?
- Which versions of HTTP are the most popular?
- What is the ratio between HTTP and HTTPS requests?

And these questions are answered by the following `varnishtop` commands:

```
varnishtop -i ReqUrl -q "VCL_call eq 'MISS'"
varnishtop -i ReqUrl -q "VCL_return eq 'pass'"
varnishtop  -i BeReqUrl -q "TTL[6] eq 'uncacheable'"
varnishtop -i BeReqUrl -q "Timestamp:Beresp[3] > 2.0"
varnishtop -i RespProtocol
varnishtop -I Begin:sess
```

We won't be showing the output for the commands where only the `ReqUrl` or `BeReqUrl` are used because that output is very predictable.

However, we am going to show some output for the last three commands.

Here's the output you get when you want to figure out what the most popular HTTP version is:

```
list length 2
   310.29 RespProtocol   HTTP/1.1
    19.56 RespProtocol   HTTP/2.0
```

As you can see, the test setup received a lot more `HTTP/1.1` requests than `HTTP/2.0` requests.

As far as *HTTP* versus *HTTPS* is concerned, you have to interpret the following output the right way:

```
list length 2

   234.25 Begin          sess 0 HTTP/1
    49.89 Begin          sess 0 PROXY
```

The `sess 0 HTTP/1` log line represents straightforward *HTTP* requests that were received via a standard *HTTP listener*. Because *HTTPS* was terminated via *Hitch* and sent over the *PROXY protocol*, the `sess 0 PROXY` log line represents *HTTPS* traffic.

Not that straightforward, admittedly. An easier way is to inject custom logging information, as illustrated below:

```
vcl 4.1;

import proxy;
import std;


sub vcl_recv {
    if(proxy.is_ssl()) {
        std.log("scheme: HTTPS");
    } else {
        std.log("scheme: HTTP");
    }
}
```

We can log the URL scheme using `std.log()` and use the `scheme` prefix for filtering.

This would result in the following `varnishtop` command:

```
varnishtop -I VCL_Log:scheme
```

And finally, we would get some sensible output:

```
list length 2

    16.49 VCL_Log        scheme: HTTPS
    11.36 VCL_Log        scheme: HTTP
```

On *Varnish Enterprise* such custom logging is even required because *native TLS* doesn't use the *PROXY protocol*. There is no way to easily distinguish *HTTP* from *HTTPS* via `Begin:sess`.

We can take our `vmod_proxy` example and refactor it for *native TLS* purposes:

```
vcl 4.1;

import tls;
import std;


sub vcl_recv {
    if(tls.is_tls()) {
        std.log("scheme: HTTPS");
    } else {
        std.log("scheme: HTTP");
    }
}
```

This would allow us to use the same `varnishtop -I VCL_Log:scheme` command and get the same type of output.

7.8.9 Running varnishncsa as a service

If you install *Varnish* via packages, not only is there a *systemd unit file* for `varnishd`, there's also one for `varnishncsa`.

It's located in `/lib/systemd/system/varnishncsa.service` and looks like this:

```
[Unit]
Description=Varnish Cache HTTP accelerator NCSA logging daemon
After=varnish.service

[Service]
RuntimeDirectory=varnishncsa
Type=forking
User=varnishlog
Group=varnish

ExecStart=/usr/bin/varnishncsa -a -w /var/log/varnish/varnishncsa.log
-D
ExecReload=/bin/kill -HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

The following `varnishncsa` command runs:

The fact that the `-D` option is used, means `varnishncsa` is daemonized to run in the background. You are basically running `varnishncsa` as a service. The service is managed by `systemd`.

However, the service is not active by default. Run the following commands to enable and start the `varnishncsa` service:

```
sudo systemctl enable varnishncsa
sudo systemctl start varnishncsa
```

You'll find the logs in `/var/log/varnish/varnishncsa.log`. The `logrotate` service will ensure that the logs are properly rotated, based on the `/etc/logrotate.d/varnish` configuration.

When log rotation is due, the `systemctl reload varnishncsa.service` will be called via `logrotate`. As you can see, the `varnishncsa.service` file handles reloads using the following command:

```
/bin/kill -HUP $MAINPID
```

And this brings us back to the `SIGHUP` signal that `varnishncsa` uses to release and reopen the log file.

Without further adjustments to the `varnishncsa.service` file, the standard log format will be used.

Please beware of using custom formats in `varnishncsa.service`. Please note that `systemd` also uses percent `%` expansion in unit files. If you're changing the format via the `-F` option, you need to double the percent signs in your format string.

For example the default format would look like this:

```
-F '%%h %%l %%u %%t "%%r" %%s %%b "%%{Referer}i" "%%{User-agent}i"'
```

Alternatively you can read the format from a file using the `-f` option in your `varnishncsa` command.

Editing the service to modify the format can be done using the following command:

```
sudo systemctl edit --full varnishncsa.service
```

After having added a `-F` or `-f` option, and after having modified the output format, don't forget to restart the service using the following command:

```
sudo systemctl restart varnishncsa.service
```

Editing the unit file via `systemctl edit` results in a *symlink* of the unit file being created in `/etc/systemd/system`. You can also edit that file directly or create the *symlink* directly instead of using `sudo systemctl edit --full varnishncsa.service`. However, don't forget to call `sudo systemctl daemon-reload` before restarting your service.

By copying the `varnishncsa.service` and giving it another name, you can run other `varnishncsa` services that may include custom formatting and *VSL queries*. However, don't forget to update `/etc/logrotate.d/varnish` with the location and log-rotation scenario for those services, otherwise you might run out of disk space.

7.8.10 Why wasn't this page served from cache?

To us, `varnishlog` is an indispensable tool.

You can set up *Varnish* and properly configure it. You can make sure your *origin* application sends the right *HTTP headers* to control the behavior of *Varnish*. You can even tailor the behavior of *Varnish* to your exact needs by writing *VCL*.

But how do you know your *content delivery strategy* is performing well? *VSL*, and more specifically, `varnishlog` can answer that question in great detail. Let's approach this using the *built-in VCL* and ask ourselves a very common question:

Why wasn't this page served from cache?

You could list all transactions that resulted in a *pass*. The following command will help you with that:

```
varnishlog -i ReqUrl -i ReqMethod -i ReqProtocol -I ReqHeader:Host \
    -I ReqHeader:Cookie -I ReqHeader:Authorization \
    -i VCL_call -i VCL_return \
    -q "VCL_call eq ‘PASS’"
```

You could also be looking at a specific URL and ask yourself the same question:

```
varnishlog -i ReqUrl -i ReqMethod -i ReqProtocol -I ReqHeader:Host \
    -I ReqHeader:Cookie -I ReqHeader:Authorization \
    -i VCL_call -i VCL_return \
    -q "ReqUrl eq ‘/’"
```

Let's cover the various *built-in VCL* scenarios, and show how the `varnishlog` output answers our question.

Remember: these decisions are made within the `vcl_recv` subroutine.

Because it was a POST request

Based on the following output, we can conclude that `http://localhost/` wasn't served from cache because it was a `POST` call:

```
*   << Request  >> 2
-   ReqMethod      POST
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   VCL_call       RECV
-   VCL_return     pass
-   VCL_call       HASH
-   VCL_return     lookup
-   VCL_call       PASS
-   VCL_return     fetch
-   VCL_call       DELIVER
-   VCL_return     deliver
```

Remember, only `GET` and `HEAD` requests are cached in the *built-in VCL*.

Because the request contained a cookie

The following output shows that the page wasn't served from cache because the request contained a *cookie*:

```
*   << Request  >> 3
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   ReqHeader      Cookie: lang=en
-   VCL_call       RECV
-   VCL_return     pass
-   VCL_call       HASH
-   VCL_return     lookup
-   VCL_call       PASS
-   VCL_return     fetch
-   VCL_call       DELIVER
-   VCL_return     deliver
```

The *built-in VCL* performs a *pass* when *cookies* appear in the request.

In situations where `regsub()`, `regsuball()`, `cookie.delete()`, `cookie.filter()`, `cookie.keep()`, `cookieplus.delete()`, or `cookieplus.keep()` are used to remove or keep specific cookies, it is possible that an unexpected cookie slipped through the cracks and caused an unexpected *pass*. *VSL* helps you figure this out.

Because an authorization header was passed

The *built-in VCL* also doesn't cache when an `Authorization` header is part of the request.

You can easily spot it in the output below:

```
*   << Request  >> 98327
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   ReqHeader      Authorization: Basic dGVzdDp0ZXN0
-   VCL_call       RECV
-   VCL_return     pass
-   VCL_call       HASH
-   VCL_return     lookup
-   VCL_call       PASS
-   VCL_return     fetch
-   VCL_call       DELIVER
-   VCL_return     deliver
```

Because we couldn't recognize the request method

There is still another reason why an object wasn't served from cache. This situation doesn't result in a *pass*, but instead results in a *pipe*.

This occurs when an unrecognized request method is used. `return(pipe);` is called from within the *VCL*, and what was supposed to be an *HTTP request* is simply treated as a *TCP byte stream*.

To track these kinds of requests, we need to slightly modify our `varnishlog` command:

```
varnishlog -c -i ReqUrl -i ReqMethod -i ReqProtocol -I ReqHeader:Host
\
    -I ReqHeader:Cookie -I ReqHeader:Authorization \
    -i VCL_call -i VCL_return \
    -q "VCL_return eq ‘pipe’"
```

As you can see `VCL_return eq ‘pipe’` is our *VSL query*, and this is the result:

```
*   << Request  >> 20
-   ReqMethod      FOO
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
-   ReqHeader      Host: localhost
-   VCL_call       RECV
-   VCL_return     pipe
-   VCL_call       HASH
-   VCL_return     lookup
```

Because our *request method* was `FOO`, the *built-in VCL* didn't exactly know what to do with that and decided to *pipe* the request directly to the backend.

7.8.11 Why wasn't this page stored in cache?

Whereas serving objects from cache is a *client-side responsibility*, storing objects in cache is a *backend responsibility*. `varnishlog` also has the capabilities of explaining why certain pages weren't stored in cache.

We will again base ourselves on the criteria of the *built-in VCL*. If you remember the flow correctly, you'll know that these decisions are made in the `vcl_backend_response` subroutine.

This is the command we will use to figure out what's going on:

```
varnishlog -g request -b -i BeReqUrl -I BerespHeader:Cache-Control \
    -I BerespHeader:Expires -I BerespHeader:Vary -I BerespHead-
er:Set-Cookie \
    -I BerespHeader:Surrogate-Control -i TTL -q "TTL[6] eq 'uncache-
able'"
```

It's quite lengthy, but here's a breakdown for you:

- Transactions are grouped by *request* (`-g request`).
- We're only displaying *backend transactions* (`-b`).
- The *backend request URL* is included in the output (`-i BeReqUrl`).
- We want to see the value of the `Cache-Control` header (`-I BerespHeader:Cache-Control`).
- We want to see the value of the `Expires` header (`-I BerespHeader:Expires`).
- We want to see the value of the `Surrogate-Control` header (`-I BerespHeader:Surrogate-Control`).
- The `Vary` header should also be displayed (`-I BerespHeader:Vary`).
- The `Set-Cookie` header is also an important part of the output (`-I BerespHeader:Set-Cookie`).
- The `TTL` tag will help us understand what *TTL* is chosen (`-i TTL`).
- That same `TTL` tag is used in the *VSL query* to only selected *uncacheable responses* (`-q "TTL[6] eq 'uncacheable'"`).

Zero TTL

The first case we're going to examine is a *zero TTL*. This means that the origin returned a `Cache-Control` header, or an `Expires` header, which set the *TTL* to zero.

These are the cases that could trigger this:

```
Cache-Control: max-age=0
Cache-Control: s-maxage=0
Expires: Thu, 1 Jan 1970 12:00:00 GMT
```

A *zero TTL* can also be set in *VCL*:

```
set beresp.ttl = 0s;
```

Let's have a look at some *VSL* output that shows this in the logs:

```
**   << BeReq    >> 24
--   BereqURL       /
--   BerespHeader   Cache-Control: max-age=0
--   TTL            RFC 0 10 0 1607071591 1607071591 1607071590 0 0
cacheable
--   TTL            VCL 0 10 3600 1607071591 cacheable
--   TTL            VCL 120 10 3600 1607071591 cacheable
--   TTL            VCL 120 10 3600 1607071591 uncacheable
```

The `Cache-Control` header has a `max-age` that is zero. This is enough to set `beresp.ttl` to zero seconds.

The first occurrence of the `TTL` tag shows this:

- The `RFC` value indicates the *TTL* was set via headers.
- The *TTL field* is `0`.
- The second to last field, which represents the `max-age` value, is also `0`.

Although it is considered cacheable at first, the *built-in VCL* will set it to *uncacheable*, which triggers *hit-for-miss* behavior. And as you can see the *TTL* is set to `120`.

When the `s-maxage` is set to zero, even though `max-age` equals `100`, the same thing happens:

```
**  << BeReq    >> 32783
--  BereqURL       /uncacheable
--  BerespHeader   Cache-Control: max-age=100 s-maxage=0
--  TTL            RFC 0 10 0 1607072077 1607072077 1607072077 0 0
cacheable
--  TTL            VCL 0 10 3600 1607072077 cacheable
--  TTL            VCL 120 10 3600 1607072077 cacheable
--  TTL            VCL 120 10 3600 1607072077 uncacheable
```

The same thing happens when the `Expires` header is set to the past, but the `Cache-Control` header has cacheable `max-age` and `s-maxage` values:

```
**  << BeReq    >> 32798
--  BereqURL       /
--  BerespHeader   Cache-Control: max-age=100 s-maxage=100
--  BerespHeader   Expires: Thu, 1 Jan 1970 12:00:00 GMT
--  TTL            RFC 0 10 0 1607072365 1607072365 1607072364 0 0
cacheable
--  TTL            VCL 0 10 3600 1607072365 cacheable
--  TTL            VCL 120 10 3600 1607072365 cacheable
--  TTL            VCL 120 10 3600 1607072365 uncacheable
```

Private, no-cache, no-store

And even if `max-age` is greater than zero, it is still possible that `Cache-Control` semantics prevent the object from being cached.

Here's an example where `private` results in the object becoming *uncacheable*:

```
**  << BeReq    >> 44
--  BereqURL       /
--  BerespHeader   Cache-Control: private, max-age=3600
--  TTL            RFC 3600 10 0 1607072499 1607072499 1607072498 0
3600 cacheable
--  TTL            VCL 3600 10 3600 1607072499 cacheable
--  TTL            VCL 120 10 3600 1607072499 cacheable
--  TTL            VCL 120 10 3600 1607072499 uncacheable
```

You see in the first occurrence of the `TTL` tag that the second-to-last field is set to `3600`. This corresponds to the `max-age` value from the `Cache-Control` header. But because of the `private` keyword, the object is going in *hit-for-miss* mode for the next *two minutes*.

Either `private`, `no-cache`, or `no-store` will cause this to happen. Here's an example where all three are used:

```
**   << BeReq    >> 32801
--   BereqURL       /
--   BerespHeader   Cache-Control: private, no-cache, no-store
--   TTL            RFC 120 10 0 1607072684 1607072684 1607072683 0 0
cacheable
--   TTL            VCL 120 10 3600 1607072684 cacheable
--   TTL            VCL 120 10 3600 1607072684 cacheable
--   TTL            VCL 120 10 3600 1607072684 uncacheable
```

Surrogate-control no-store

The `Surrogate-Control` header is also an important one. When it contains `no-store`, it takes precedence over any valid `Cache-Control` header in the *built-in VCL*.

You can see for yourself in the following *VSL output*:

```
**   << BeReq    >> 32810
--   BereqURL       /
--   BerespHeader   Cache-Control: public ,max-age=100
--   BerespHeader   Surrogate-Control: no-store
--   TTL            RFC 100 10 0 1607073237 1607073237 1607073237 0
100 cacheable
--   TTL            VCL 100 10 3600 1607073237 cacheable
--   TTL            VCL 120 10 3600 1607073237 cacheable
--   TTL            VCL 120 10 3600 1607073237 uncacheable
```

Even though the `RFC 100 10 0 1607073237 1607073237 1607073237 0 100 cacheable` line indicates that the response is cacheable because of the `Cache-Control` value, it eventually is deemed *uncacheable* because of the `Surrogate-Control: no-store` header.

Setting a cookie

Setting a *cookie* implies a state change. The *built-in VCL* takes this into account and prevents such objects from being stored in cache.

As you can see in the following *VSL output*, despite the `Cache-Control` header that resulted in a *TTL* of *100 seconds*, the object is still considered *uncacheable*. The `Set-Cookie` header is to blame for that:

```
**   << BeReq    >> 59
--   BereqURL       /1
--   BerespHeader   Cache-Control: public ,max-age=100
--   BerespHeader   Set-Cookie: id=098f6bcd4621d373cade4e832627b4f6
--   TTL            RFC 100 10 0 1607073445 1607073445 1607073444 0
100 cacheable
--   TTL            VCL 100 10 3600 1607073445 cacheable
--   TTL            VCL 120 10 3600 1607073445 cacheable
--   TTL            VCL 120 10 3600 1607073445 uncacheable
```

Wildcard variations

There's still one condition that would trigger *hit-for-miss* behavior that we need to cover: *wildcard variations*.

As you can see in the output below, the response contains a `Vary: *` header:

```
**   << BeReq    >> 32813
--   BereqURL       /12
--   BerespHeader   Cache-Control: public ,max-age=100
--   BerespHeader   Vary: *
--   TTL            RFC 100 10 0 1607074172 1607074172 1607074172 0
100 cacheable
--   TTL            VCL 100 10 3600 1607074172 cacheable
--   TTL            VCL 120 10 3600 1607074172 cacheable
--   TTL            VCL 120 10 3600 1607074172 uncacheable
```

By setting a variation on every header, which is represented by the asterisk, your hit rate is going to fall off a cliff. The *built-in VCL* considers this to be an uncacheable case, which is reflected in the last occurrence of the `TTL` tag.

7.8.12 The significance of VSL

We've reached the end of this section, and you must admit that it was quite in-depth. That is because *VSL* should be a crucial part of your debugging strategy.

Although monitoring counters via `varnishstat` can give you an indication of what is going on, `varnishlog`, `varnishncsa`, and `varnishtop` allow you to test some of these assumptions or conclusions.

Please also use *VSL* when setting up *Varnish* and when writing *VCL*. Sometimes your *origin* behaves differently than you would expect. The logs will help you figure out what's going on and may result in *VCL* changes.

And finally, we would advise anyone to install *Varnish* and the *VSL* tools, even if you're not planning to cache any *HTTP responses*. Just the fact of having such an in-depth tool with so many filtering capabilities is an asset on its own.

7.9 Security

As the significance of online services increases, and as security risks increase at the same time, it is crucial to have the necessary security measures in place.

Widely covered vulnerabilities like *Heartbleed*, *Shellshock*, *Spectre*, and *Meltdown* were a wakeup call for the IT industry and changed the security landscape.

In this section we'll cover security from two angles:

- *Prevention*: how do we reduce attack vectors?
- *Mitigation*: how do we reduce the damage if we still manage to get hacked?

Because *Varnish* operates at *the edge*, it is our first line of defense, but also the first component that will be under attack.

Although caches are designed to store large amounts of data in a tightly packed space, and although these systems prioritize performance, *Varnish* itself does an exceptional job in defensive coding practices, secure design, and maintaining cache integrity.

But that doesn't mean we shouldn't pay attention to security and potential risks. Let's look at how we can prevent hacking and mitigate damage.

7.9.1 Firewalling

The very first thing we do is to shut down all ports that are not essential. This is a preventive measure.

Varnish will typically operate on port 80. If *native-TLS* is active, port 443 also needs to be accessible. The same applies if *Hitch* is used for TLS termination.

There are situations where *Varnish* sits behind a load balancer. In that case, the load balancer will be exposed to the outside world, and *Varnish* isn't.

Although port 80 and 443 will be exposed to the outside world, there is of course the access to the *Varnish CLI*.

The *Varnish CLI*, which runs on port 6082 by default, should only be accessed by IP addresses or IP ranges that are entitled to access it.

In most cases these will be private IP addresses or ranges that aren't accessible via the internet. In that case it makes sense to set the `-T` parameter to only listen on an private IP address within the range of the management network.

7.9.2 Cache encryption

In the unlikely event that someone can hack the `varnishd` process, cached data can be accessed, and maybe even modified. Depending on the sensitivity of that data, this might result in a serious security risk.

A possible mitigation strategy for *Varnish Enterprise* users is to use *Total Encryption*.

Total Encryption is a *Varnish Enterprise* feature, written in *VCL*, that leverages `vmod_crypto`.

Using *Total Encryption* for non-persistent memory caches only requires the following include:

```
include "total-encryption/random_key.vcl";
```

The files you include are automatically shipped with *Varnish Enterprise*.

Objects are encrypted using an *AES256 encryption cipher* with a dual-key algorithm for extra security.

- The first key is *128-bit randomly generated number* that is stored in kernel space for the duration of `varnishd`'s lifetime.
- The second key contains the *request hash* and isn't stored anywhere.

These two keys are used to create an *HMAC signature* that represents our master key. The random number is the key of our *HMAC signing object*, and the request hash is the value that is signed.

The *HMAC signature* is generated using the *Linux Kernel Crypto API*. This means that values are never stored in user space and are kept inside the Linux kernel. When `varnishd` restarts, a new master key is generated.

Once we have the master key, we can start encrypting. Behind the scenes the `crypto.aes_encrypt_response()` and `crypto.aes_decrypt_response()` functions are used to encrypt and decrypt content. Encryption happens in a `vcl_backend_response` hook, whereas the decryption happens in a `vcl_deliver` hook.

Not only does our *AES256* encryption use our dual-key algorithm, we also add a randomly generated salt for extra security.

Total Encryption doesn't really care whether or not you passed the right master key. It uses whatever key is presented, and if it cannot successfully decrypt the content, garbage is returned. So even if you tamper with the settings, the only way to successfully decrypt an object is if you know the request hash, the master key, and the salt.

And even if you succeed, you can only decrypt that single object because every object uses a different key.

Encrypting persisted cache objects

For persisted objects, our dual-key algorithm is implemented slightly differently.

Because the persisted cache can outlive the `varnishd` process, we cannot rely on the random key to still be the same for that object.

The solution is to use a *local secret key* that is stored on disk. We still use the *request hash* for the second key.

Here's a safe way to generate the local key:

```
$ cat /dev/urandom | head -c 1024 > /etc/varnish/disk_secret
$ sudo chmod 600 /etc/varnish/disk_secret
$ sudo chown root: /etc/varnish/disk_secret
```

As you can see, the key is long enough to be secure, and the permissions are tightly locked down. The end result is the `/etc/varnish/disk_secret` file.

Varnish Enterprise uses the `-E` runtime parameter to take in the secret key that is used by the `crypto.secret()` function to expose it to *VCL*.

Again, the key is not stored inside the `varnishd` process but is kept in the *Linux kernel*.

Here's an example of the `-E` runtime parameter:

```
varnishd -a :80 -f /etc/varnish/default.vcl -E /et/varnish/disk_se-
cret
```

Whereas memory caches include the `total-encryption/random_key.vcl` file, this is how persisted caches should enable *Total Encryption*:

```
include "total-encryption/secret_key.vcl";
```

The rest of the behavior is identical and will ensure that persisted objects can also be encrypted and decrypted.

Performance impact

Varnish Total Encryption performance is on par with any other *AES* implementation. *AES* calculations are hardware accelerated and are quite *CPU-intensive*.

We performed some benchmarks, and here are some performance results with and without *Total Encryption*:

	Requests	Bandwidth	Response time
Unencrypted	23068	17.61 Gbit	0.084 ms
Total Encryption	11353	8.68 Gbit	0.135 ms
Overhead	50.78%	50.77%	61.68%

As you can derive from the table, there is a 50% performance overhead when using *Total Encryption*. These tests were run on a *four-core server* with *100 KB objects*.

Because the performance decrease is related to the CPU, adding more CPUs will bring your performance back to original levels.

Skipping encryption

Primarily because of the performance overhead, there might be situations where you don't want to encrypt certain objects.

The `crypto.aes_skip_response()` will make sure the current object is not encrypted. The example below uses this function to skip encryption on video files:

```
vcl 4.1;

sub vcl_backend_response {
    if (beresp.http.Content-Type ~ "video") {
        crypto.aes_skip_response();
    }
}
```

If you're certain that some objects don't contain any sensitive data, and if you suspect these objects are quite big, skipping them might be a good decision.

Some objects are automatically skipped: if it turns out the object contains an *HTTP 304* response, it will be skipped. Because if you remember, an *HTTP 304* response has no response body, so there's no need to encrypt it.

Choosing an alternate encryption cipher

The standard *AES* implementation uses *cipher block chaining (CBC)*. If you want to switch to *propagating cipher block chaining (PCBC)*, you can set it by modifying the `algorithm` configuration setting in the `te_opts` key-value store.

Here's how you can do this:

```
sub vcl_init {
    te_opts.set("algorithm", "pcbc(aes)");
}
```

Header encryption

Although *Total Encryption* encrypts the reponse body of an *HTTP response*, it doesn't encrypt the headers.

`vmod_crypto` does have the required methods to achieve this. However, you'll have to encrypt each header manually, and separately, as you can see in the example below:

```
sub vcl_backend_response {
    set beresp.http.Content-Type = crypto.hex_encode(crypto.aes_en-
crypt(beresp.http.Content-Type));
}

sub vcl_deliver {
    if (resp.http.Content-Type != "") {
        set resp.http.Content-Type = crypto.aes_decrypt(crypto.hex_
decode(resp.http.Content-Type));
    }
}
```

7.9.3 Jailing

Varnish uses jails to reduce the privileges of the *Varnish* processes.

Usually, the `varnishd` process will be run with root privileges. It uses these privileges to load the files it needs for its operations.

Once that has happened, the jailing mechanism kicks in, and `varnishd` switches to an alternative user. This is the `varnish` user by default.

The worker process that is spawned will run as the `vcache` user.

It is possible to change these values via the `-j` runtime parameter.

Here's an example where the management process uses the `varnish-mgt` user, and the worker process uses the `varnish-wrk` user:

```
varnishd -a :80 -f /etc/varnish/default.vcl -j unix,user=var-
nish-mgt,workuser=varnishwrk
```

It is even possible to define a group to which the `varnishd` process and subprocesses belong. This is done using the `ccgroup` configuration option that is also part of the `-j` runtime parameter.

Here's an example:

```
varnishd -a :80 -f /etc/varnish/default.vcl \
    -j unix,user=varnish-mgtccgroup=varnish-grp,workuser=varnish-wrk
```

7.9.4 Making runtime parameters read-only

`varnishd` parameters that are set via the `-p` option can be overridden using the `param.set` command in `varnishadm`.

Some of these parameters may result in *privilege escalation*. This can be especially dangerous if *remote CLI access* is available, and the CLI client gets compromised.

The `-r` option for `varnishd` can make certain parameters read-only.

Here's an example where some sensitive runtime parameters are made read-only:

```
varnishd -a :80 -f /etc/varnish/default.vcl -r "cc_command, vcc_al-
low_inline_c, vmod_path"
```

When we then try to change `cc_command` via `varnishadm`, we get the following message:

```
$ varnishadm param.set cc_command "bla"
parameter "cc_command" is protected.
Command failed with error code 107
```

7.9.5 VCL security

When you perform tasks in *VCL* that are restricted to authorized hosts or users, you should write security logic in your *VCL file*.

We've already covered this in *chapter 6* when we talked about purging and banning.

Here's the very first example we used in that chapter, and it contains an *ACL* to prohibit unauthorized access:

```
vcl 4.1;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        return (purge);
    }
}
```

There might even be other parts of your web platform that should only be accessible for specific IP addresses, ranges, or hostnames.

Another way to secure your *VCL*, or maybe even an additional way, is to add an authentication layer. In this case *Varnish* would serve as an authentication gateway.

We won't go into much detail about this because it will be covered in the next chapter. Let's just throw in a simple example where basic authentication is used on top of the *ACL* to protect purges:

```
vcl 4.1;

acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(405));
        }
        if (! req.http.Authorization ~ "Basic Zm9vOmJhcg==") {
            return(synth(401, "Authentication required"));
        }
        unset req.http.Authorization;
```

```
            return (purge);
        }
}

sub vcl_synth {
    if (resp.status == 401) {
        set resp.status = 401;
        set resp.http.WWW-Authenticate = "Basic";
        return(deliver);
    }
}
```

So not only does the client need to execute the purge from `localhost` or the `192.168.55.0/24` IP range, the client also needs to log in with username `foo` and password `bar`.

But again: more about authentication in the next chapter.

7.9.6 TLS

Remember the *Heartbleed* security vulnerability? The bug in the *OpenSSL* library allowed memory to be leaked and exposed sensitive data.

Although this vulnerability should no longer affect software that uses the updated *OpenSSL* version, it should serve as a warning. *Varnish Enterprise* uses *OpenSSL* for its native-TLS feature. *Hitch* also uses *OpenSSL*.

Although we really have no reason to suspect similar vulnerabilities to be present, we can put mitigating measures in place by splitting up caching and cryptography into separate services.

It sounds quite mysterious, but the truth is that it just involves using *Hitch* because *Hitch* is a separate service that runs under a different user.

Here's an example where the socket is placed under `/var/run/varnish.sock`, owned by the `varnish` user and the `varnish` group. The file has `660` permissions, which only grants read and write access to the `varnish` user and users that are in the `varnish` group:

```
varnishd -a uds=/var/run/varnish.sock,PROXY,user=varnish,group=var-
nish,mode=660 \
    -a http=:80 -f /etc/varnish/default.vcl
```

Just make sure the following *Hitch* configuration directives are set:

```
backend = "/var/run/varnish.sock"
user = "hitch"
group = "varnish"
write-proxy-v2 = on
```

The fact that the `hitch` process is owned by the `varnish` group is what allows it to access `/var/run/varnish.sock`.

In the unlikely event that your *Hitch* setup is hacked, only this part of the memory would leak. *Varnish* itself, which is a separate process, running under a separate user, would remain secure.

7.9.7 Cache busting

Cache busting is a type of attack that involves deliberately causing cache misses to bring down the origin server.

It either happens by calling random URLs or by attaching random query strings to an otherwise cached object.

Regardless of the attack specifics, the goal is to send as much traffic to the origin as possible in an attempt to bring it down.

There are measures we can take to prevent certain types of cache busting as well as measures to mitigate the impact of cache busting.

Let's have a look at the various options:

Query string filtering

Quite often, cache busting attacks use random query string parameters to cause cache misses. While we cannot completely prevent this from happening, we can at least make sure that we only allow the query string parameters we need.

To enforce this, we are using `vmod_urlplus`, an *enterprise-only VMOD* that has the capability of throwing out unwanted query string parameters.

Imagine that the attacker calls tens of thousands of URLs that look like this:

```
http://example.com/?DA80F1C6-2244-4F48-82FF-807445621783
http://example.com/?59FEF405-3292-4326-A214-53A5681D3E24
http://example.com/?BFD51681-CFE0-4C4B-A276-FB638F5FCB82
http://example.com/?DE5B3B6D-AF08-435D-B361-C5235460418E
http://example.com/?FE0B8B92-E163-4276-B12B-AF9A2B355ED6
http://example.com/?B4D05958-D0A9-4554-A64D-63D6FD4F16C5
http://example.com/?3F3915C5-97B5-4C03-ACB8-BF4FCA63D783
http://example.com/?AF86F196-26D6-4FFC-8A2D-7B5040F6B903
http://example.com/?EA72A80C-DDDB-45AF-BCE1-19357C8484DA
```

The `urlplus.keep()` and `urlplus.keep_regex()` functions will help us get rid of this garbage while still keeping important query string parameters. Here's the *VCL code*:

```
vcl 4.1;

import urlplus;

sub vcl_recv {
    urlplus.query_keep("id");
    urlplus.query_keep("sort");
    urlplus.query_keep_regex("product_*");
    urlplus.write();
}
```

This example will only keep the following query string parameters while removing all others:

- `id`
- `sort`
- All query string parameters that start with `product_`

Once the filtering is complete, `urlplus.write()` will not only write back the value to `req.url`, it will also sort the query string alphabetically.

The sorting feature is great because it prevents cache busting when the attacker reorders the query string parameters.

With this *VCL* code in place for query string filtering, we could call `/?id=1&-foo=bar&sort=asc&product_category=shoes&xyz=123` and end up with the following output:

```
varnishlog -g request -i ReqUrl
*   << Request  >> 3964951
-   ReqURL         /?id=1&foo=bar&sort=asc&product_catego-
ry=shoes&xyz=123
-   ReqURL         /?foo=bar&id=1&product_category=shoes&sort=as-
c&xyz=123
-   ReqURL         /?id=1&product_category=shoes&sort=asc
**  << BeReq    >> 3964952
```

- The first `ReqURL` tag displays the input URL.
- The second `ReqURL` tag shows the filtered version.
- The third `ReqURL` tag returns the sorted version.

If you're not using *Varnish Enterprise*, you can achieve the same result with the `regsuball()` function, but it will require writing potentially complex regular expressions.

You may also remember the `std.querystring()` function. This function is readily available in *Varnish Cache* and will at least take care of the query string sorting.

Don't forget that query string filtering only works for names, not for values. You can still add random values to the `id` and cause cache busting.

If you want to protect the values of your query string parameters, you can check their values, as illustrated here:

```
if(urlplus.query_get("id") !~ "^[0-9]{1,9}$") {
    return(synth(400));
}
```

So at least you narrow the values of `id` down to numeric ones. But this still leaves us with nine digits to abuse, which might be enough to take down the origin.

Max connections

It's safe to say that query string filtering doesn't offer a foolproof solution to prevent cache busting.

If we can't fully prevent this from happening, we can focus on reducing the impact.

By setting the `.max_connections` backend setting, we can control the maximum number of open connections to the origin server.

Here's an example where we allow a maximum of 100 connections to the origin server:

```
vcl 4.1;

backend default {
    .host = "origin.example.com";
    .port = "80";
    .max_connections = 100;
}
```

Setting `.max_connections` is always a good idea, not just to prevent attacks. But it is important to know that once the limit is reached, requests will return an *HTTP 503* error because the backend is currently not available to those requests.

Unfortunately this is not an elegant solution because you also punish regular visitors. One could even say that the *HTTP 503* is just as bad as an outage.

Backend throttling

A better mitigation strategy is to punish the culprit. We can use the `vmod_vsthrottle` to make this happen.

Previous `vmod_vsthrottle` examples featured rate limiting at the request level. This prevents users from sending too many requests within a given timeframe.

In this case, we're moving the rate-limiting logic to the backend side of *Varnish*: we'll temporarily block access to *Varnish* for users that have caused too many backend requests within a given timeframe.

Here's an example:

```
vcl 4.1;

import vsthrottle;

sub vcl_backend_fetch {
    if (vsthrottle.is_denied(client.ip, 100, 1s, 1m)) {
        return(error(429, "Too Many Requests"));
    }
}
```

If the client, identified by its client IP address makes more than 100 requests per second that result in a cache miss, access is prohibited for one minute.

Although `vmod_vsthrottle` is packaged with *Varnish Enterprise*, it is an open source module that is part of *the Varnish Software VMOD collection*. We talked about it in *chapter 5* in case you forgot.

7.9.8 Slowloris attacks

Slowloris attacks are *denial of service* attacks that hold the connection open as long as possible. The goal is to exhaust all available connections and cause new, valid connections to be refused.

Keeping the connection open is not good enough. *Varnish* has a `timeout_idle` runtime parameter with a default value of *five seconds*, which closes the client connection if it has been idle for five seconds.

Slowloris attacks are much smarter than that: they actually send partial requests, adding data as slowly as possible, to prevent the `timeout_idle` from being triggered.

By tuning the `idle_send_timeout`, you can control how long *Varnish* waits in between individual pieces of received data. The default value is *60 seconds*, which means *Varnish* is willing to wait up to one minute in between every line of data being sent.

Luckily, there's also the `send_timeout` runtime parameter with a standard value of *600 seconds*. This parameter represents the total timeout. Basically a *last byte timeout* for the request.

If you're suffering from *slowloris* attacks, you can tune these settings to mitigate the impact.

7.9.9 Web application firewall

One can say that with the power of *VCL*, and the way this allows you to intelligently block unwanted requests, *Varnish* is really also a *web application firewall (WAF)*.

However, writing the *VCL*, doing the request inspection, and making sure you're prepared for the next *zero-day exploit*, can be a lot of work.

To make things easier, and to transform *Varnish* into an actual *WAF*, *Varnish Enterprise* offers a *WAF add-on* that leverages the *ModSecurity* library.

Installing the Varnish WAF

If you have the right *Varnish Enterprise* subscription, you'll have access to the package repository that contains the `varnish-plus-waf` package.

The example below installs that package along with *Varnish Enterprise* itself. This example is targeted at *Red Hat*, *CentOS*, and *Fedora* systems:

```
sudo yum install varnish-plus varnish-plus-waf
```

If you're on a *Debian* or *Ubuntu* system, you'll use the following command:

```
sudo apt install varnish-plus varnish-plus-waf
```

The *ModSecurity* library has ruleset definitions that are stored in separate files. You can define your own rules, but you can also download the *OWASP Core Rule Set (OWASP CRS)*.

Here's how you download these rules to your *Varnish* server:

```
sudo get_owasp_crs
```

The result is that a collection of files is placed in `/etc/varnish/modsec/owasp-crs-{VERSION_NUMBER}`, which you can then load into `vmod_waf`.

In this case, this leads to the following result:

```
OWASP CRS VERSION v3.1.1 installed to /etc/varnish/modsec/owasp-
crs-v3.1.1
```

The `vmod_waf` API, and the complexity of the *WAF* is nicely abstracted by the `include "waf.vcl"` include.

Here's how you can easily enable the *WAF*:

```
vcl 4.1;

include "waf.vcl";

sub vcl_init {
    varnish_waf.add_files("/etc/varnish/modsec/modsecurity.conf");
    varnish_waf.add_files("/etc/varnish/modsec/owasp-crs-v3.1.1/
crs-setup.conf");
    varnish_waf.add_files("/etc/varnish/modsec/owasp-crs-v3.1.1/
rules/*.conf");
}
```

The `varnish_waf.add_files()` methods will load the various rulesets for the *WAF* to use.

Here's an extract of the files inside the `rules` directory:

```
REQUEST-900-EXCLUSION-RULES-BEFORE-CRS.conf.example
REQUEST-901-INITIALIZATION.conf
REQUEST-903.9001-DRUPAL-EXCLUSION-RULES.conf
REQUEST-903.9002-WORDPRESS-EXCLUSION-RULES.conf
REQUEST-903.9003-NEXTCLOUD-EXCLUSION-RULES.conf
REQUEST-903.9004-DOKUWIKI-EXCLUSION-RULES.conf
REQUEST-903.9005-CPANEL-EXCLUSION-RULES.conf
REQUEST-905-COMMON-EXCEPTIONS.conf
REQUEST-910-IP-REPUTATION.conf
REQUEST-911-METHOD-ENFORCEMENT.conf
REQUEST-912-DOS-PROTECTION.conf
REQUEST-913-SCANNER-DETECTION.conf
REQUEST-920-PROTOCOL-ENFORCEMENT.conf
REQUEST-921-PROTOCOL-ATTACK.conf
REQUEST-930-APPLICATION-ATTACK-LFI.conf
REQUEST-931-APPLICATION-ATTACK-RFI.conf
REQUEST-932-APPLICATION-ATTACK-RCE.conf
REQUEST-933-APPLICATION-ATTACK-PHP.conf
REQUEST-941-APPLICATION-ATTACK-XSS.conf
REQUEST-942-APPLICATION-ATTACK-SQLI.conf
REQUEST-943-APPLICATION-ATTACK-SESSION-FIXATION.conf
REQUEST-944-APPLICATION-ATTACK-JAVA.conf
REQUEST-949-BLOCKING-EVALUATION.conf
RESPONSE-950-DATA-LEAKAGES.conf
RESPONSE-951-DATA-LEAKAGES-SQL.conf
RESPONSE-952-DATA-LEAKAGES-JAVA.conf
RESPONSE-953-DATA-LEAKAGES-PHP.conf
RESPONSE-954-DATA-LEAKAGES-IIS.conf
RESPONSE-959-BLOCKING-EVALUATION.conf
RESPONSE-980-CORRELATION.conf
RESPONSE-999-EXCLUSION-RULES-AFTER-CRS.conf.example
```

SQL injections are a common way to take advantage of an application that uses a *SQL database*. When input parameters are parsed in the *SQL statement*, and the input is poorly secured, an attacker can inject malicious input in an attempt to retrieve data or to modify the database.

Inside `REQUEST-942-APPLICATION-ATTACK-SQLI.conf` there are a variety of SQL-related rules; here's a specific one that detects attempts to run `sleep()` statements:

```
SecRule REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|REQUEST_COOKIES_
NAMES|ARGS_NAMES|ARGS|XML:/* "@rx (?i:sleep\(\s*?\d*?\s*?\)|bench-
mark\(.*?\,.*?\))" \
    "id:942160,\
    phase:2,\
    block,\
    capture,\
    t:none,t:urlDecodeUni,\
    msg:'Detects blind sqli tests using sleep() or benchmark().',\
    logdata:'Matched Data: %{TX.0} found within %{MATCHED_VAR_NAME}:
%{MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'platform-multi',\
    tag:'attack-sqli',\
    tag:'OWASP_CRS/WEB_ATTACK/SQL_INJECTION',\
    ver:'OWASP_CRS/3.1.1',\
    severity:'CRITICAL',\
    setvar:'tx.msg=%{rule.msg}',\
    setvar:'tx.sql_injection_score=+%{tx.critical_anomaly_score}',\
    setvar:'tx.anomaly_score_pl1=+%{tx.critical_anomaly_score}',\
    setvar:'tx.%{rule.id}-OWASP_CRS/WEB_ATTACK/SQLI-%{MATCHED_VAR_
NAME}=%{tx.0}'"
```

However, by default the *WAF* won't block these kinds of requests because the `SecRuleEngine DetectionOnly` setting doesn't allow this.

By setting `SecRuleEngine on` in `/etc/varnish/modsec/modsecurity.conf`, requests matching any of the *ModSecurity* rules will be blocked.

Here's the malicious request that injects `sleep(10)` into the request body of a *POST request*:

```
curl -XPOST -d "sleep(10)" http://example.com
```

The *VSL* `WAF` tag reports the following:

```
varnishlog -g request -i WAF
*   << Request  >> 5
**  << BeReq    >> 6
--  WAF            proto: / POST HTTP/1.1
--  WAF            ReqHeaders: 8
--  WAF            ReqBody: 9
--  WAF            LOG: [client 127.0.0.1] ModSecurity: Access de-
nied with code 403 (phase 2). Matched "Operator `Ge' with parameter
`5' against variable `TX:ANOMALY_SCORE' (Value: `10' ) [file "/etc/
varnish/modsec/owasp-crs-v3.1.1/rules/REQUEST-949-BLOCKING-EVALUA-
TION.conf"] [line "80"] [id "949110"] [rev ""] [msg "Inbound Anomaly
Score Exceeded (Total Score: 10)"] [data ""] [severity "2"] [ver ""]
[maturity "0"] [accuracy "0"] [tag "application-multi"] [tag "lan-
guage-multi"] [tag "platform-multi"] [tag "attack-generic"] [hostname
"127.0.0.1"] [uri "/"] [unique_id "161070346090.720057"] [ref ""]
```

Because we matched the security rule, the anomaly score increased and exceeded the threshold. That's why we get the `Inbound Anomaly Score Exceeded` message in the output.

In the end, we received an `HTTP/1.1 403 Forbidden` response, preventing us from impacting a potentially vulnerable origin server.

When using the *Varnish WAF* it is advisable to set the `thread_pool_stack` runtime parameter to *96 KB*. This can be done by adding `-p thread_pool_stack=96k` to `varnishd`.

7.10 Tuning Varnish

Out-of-the-box *Varnish* performs exceptionally well. But based on the available server resources, based on your traffic patterns, and other criteria, you might not get the most out of *Varnish*.

It's also entirely possible that the default settings are too taxing on your system.

Either way, there are dozens of parameters you can tune. The goal is to strike a balance between making efficient use of the available resources and protecting your server from excessive load.

7.10.1 Threading settings

When we talk about getting the most out of *Varnish*, this usually means increasing the amount of simultaneous requests.

The threading settings are the best way to tune the concurrency of *Varnish*. As mentioned in the *Under The Hood* section in *chapter 1*: `thread_pool_min` and `thread_pool_max` are the settings that control how many threads are in the thread pool.

As a quick reminder: instead of creating threads on demand, a couple of thread pools are initialized that contain a certain number of threads that are ready to use. Creating and destroying threads can create overhead and cause a slight delay.

By default there are two thread pools, which can be changed by modifying the `thread_pools` parameter. But benchmarks have shown no significant improvement by changing the value.

When `varnishd` starts, *200 threads* are created in advance: *100 threads per thread pool*. The `thread_pool_min` parameter can be used to tune this number.

Growing the thread pools

When the thread pools are running out of available threads, *Varnish* will grow the pools until `thread_pool_max` is reached. Growing the pools is also somewhat resource intensive.

By increasing `thread_pool_min`, it's easier to cope with an onslaught of incoming traffic upon restart. This is common when *Varnish* sits behind a load balancer and is suddenly added to the rotation.

On the other hand, starting varnishd with too many threads will have an impact on the resource consumption of your server.

A good indicator is the MAIN.threads counter in varnishstat. It lets you know how many threads are currently active. You can correlate this number to your resource usage, and it helps you establish a baseline.

The thread_pool_min parameter should at least be the number of threads in use for an average traffic pattern.

When the MAIN.threads_limited counter increases, it means the thread pools ran out of available threads. If there is room to grow the pools, *Varnish* will add threads. If not, tasks will be queued until a thread is available.

The MAIN.threads_limited counter might increase early on when the MAIN.threads counter reaches the thread_pool_min threshold. As the pools are grown, it won't occur that much. But when MAIN.threads reaches the thread_pool_max value, the change rate of the MAIN.threads_limited counter can indicate problematic behavior.

The thread queue doesn't have an infinite size: each thread pool has a queue limit of *20 tasks*. This is configurable via the thread_queue_limit parameter. When the queue is full, any new task or request will be dropped.

So when the MAIN.threads_limited counter increases, and the MAIN.sess_dropped or MAIN.req_dropped counters are increasing as well, know that the queue is full and sessions/streams are being dropped.

The MAIN.sess_dropped counter refers to HTTP/1.1 sessions being dropped, whereas MAIN.req_dropped refers to HTTP/2 streams being dropped.

You can choose to increase the thread_queue_limit, which will allow more tasks to be queued. Unless resources are too tight, you really want to increase thread_pool_min because it will make your system more responsive.

Shrinking the thread pools

When a thread has been idle for 300 seconds, *Varnish* will clean it up. This is controlled by the thread_pool_timeout parameter. The MAIN.threads counter will reflect this.

This means *Varnish* will automatically shrink the thread pools based on demand, but with a delay. But if increased server load, caused by varnishd worker threads, is too much for your system, you should decrease thread_pool_max to an acceptable level.

If you believe *Varnish* needs to clean up idle threads quicker, you can reduce the `thread_pool_timeout`. But remember: destroying threads also consumes resources.

Another factor that will impact server load is the worker stack size. This is stack space that is consumed by every worker thread. By limiting the stack size, we manage to reduce the memory footprint of *Varnish* on the system. The size is configurable via the `thread_pool_stack` parameter. The default stack size in *Varnish* is * *48 KB*. The default process stack size on *Linux* is typically multiple orders of magnitude larger than the stack sizes we use in *Varnish*.

Stack space is typically consumed by third-party libraries that are used by *Varnish*. `libpcre`, the library to run *Perl Compatible Regular Expressions*, can consume quite a bit of stack space. If you write very complicated regular expressions in *VCL*, you might even cause a *stack overflow*.

When a stack overflow happens, you should increase the value of `thread_pool_stack`. But this, in its turn, will have a direct impact on resource consumption because the worker stack size is per thread.

If you set your worker stack size to *100 KB* and you have *5000 threads* in two thread pools, this will consume almost *1 GB* of memory. So be careful, and consider reducing `thread_pool_max` when this would be too taxing on your system.

7.10.2 Client-side timeouts

Varnish has some client-side timeouts that can be configured, which can improve the overall experience.

Most of these settings have already been discussed in the *security* section of this chapter, as they can be used to mitigate *denial of service (DoS)* attacks.

The `timeout_idle` parameter is one of these. It's a sort of *keep-alive timeout* that defines how long a connection remains open after a request. If no new pipelined request is received on the connection within five seconds, the connection is closed.

The `idle_send_timeout` parameter defines how long *Varnish* is willing to wait for the next bytes, after having already received data. This is a typical *between-bytes timeout*.

And then there's also the `send_timeout` parameter, which acts as a *last-byte timeout*.

From a *DoS* perspective these settings can help you prevent *slowloris* attacks, as mentioned earlier.

From a performance point of view, these settings can also be used to improve the end-user experience. If your *Varnish* servers sit behind a set of load balancers, it makes sense to increase `timeout_idle` because you know they are the only devices that are directly connecting to *Varnish*, and they are most probably going to reuse their connections with *Varnish*.

If you're handling large volumes of data that are processed by potentially slow clients, you can also increase the `send_timeout` value.

7.10.3 Backend timeouts

For requests that cannot be served from cache, a backend connection is made to the origin server, which acts as the *source of truth*.

If your backend is slow, or the connection is unreliable, backend connections might be left open for too long. It is also possible that the connection is closed while data is still being sent.

In order to strike the right balance, *Varnish* offers a set of backend timeouts. You should already be familiar with the settings, as they are configurable in your *VCL backend definition*.

The backend timeouts you can tune are the following ones:

- `connect_timeout`: the amount of time *Varnish* is willing to wait for the backend to accept the connection
- `first_byte_timeout`: the timeout for receiving the first byte from the origin
- `between_bytes_timeout`: the amount of time we are willing to wait in between receiving bytes from the backend

Here's a quick reminder on how to configure this in *VCL*:

```
vcl 4.1;

backend default {
    .host = "origin.example.com";
    .port = "80";
    .connect_timeout = "10s";
    .first_byte_timeout = "90s";
    .between_bytes_timeout = "5s";
}
```

These settings are of course also available as `varnishd` runtime parameters, but it is important to know that the values in *VCL* are on a per-backend basis, and take precedence over the runtime parameters.

These parameters can also be specified on a per-request basis, using `bereq.connect_timeout` or `bereq.first_byte_timeout` from `VCL_backend_fetch` in VCL.

The `backend_idle_timeout` parameter is not configurable in *VCL*, defaults to *60 seconds*, and defines how long a backend connection can be idle before *Varnish* closes it.

7.10.4 Workspace settings

You might remember the concept of *workspaces* from the *Under The Hood* section in *chapter 1*. Workspaces is a concept that is used to lessen the strain on the memory allocator where a chunk of memory is allocated for a specific task in *Varnish*.

Whereas the *stack space* is an operating system concept, the *workspace* is a *Varnish*-specific concept. The workspace memory is used for request and response parsing, for *VCL* storage, and also for any *VMOD* requiring memory space to store data.

There are different kinds of workspaces, and each of them can be tuned. When a workspace overflow occurs, this means the transactions couldn't allocate enough memory to perform their tasks.

- The `workspace_client` parameter, with a default value of *64 KB*, is used to limit memory allocation for HTTP request handling.
- The `workspace_backend` parameter, which also has a default value of *64 KB*, sets the amount of memory that can be used during backend processing.
- The `workspace_session` parameter limits the size of workspace memory used to establish the *TCP* connection to *Varnish*. The default value is *0.5 KB*.

When a task consumes more memory than allowed in one of the specific workspace contexts, the transaction is aborted, and an *HTTP 503* response is returned. When a `workspace_session` overflow occurs, the connection will be closed.

It is always possible to increase the size of the various workspaces. Memory consumption depends on what happens in *VCL*, but also depends on the input *Varnish* receives from clients and backends.

A better solution is to optimize your *VCL*, or reduce the size and the amount of headers that are sent by the backend. But sometimes, you have no control over this, or no way to significantly reduce memory consumption. In that case, increasing `workspace_client` or `workspace_backend` is your best move.

Luckily there are ways to monitor *workspace overflow*. These workspaces have a `varnishstat` overflow counter:

- `MAIN.ws_client_overflow`
- `MAIN.ws_backend_overflow`
- `MAIN.ws_session_overflow`

When these counters start increasing, don't blindly increase the workspace size. Instead, have a look at your logs, see which transactions cause the overflow, and try to figure out if you can optimize that part of your *VCL* to avoid the overflows in the future.

As always, `varnishstat` and `varnishlog` will be the tools you need to figure out what is going on before deciding to increase the size of the workspaces.

7.10.5 HTTP limits

HTTP requests and responses are parsed by *Varnish*. As mentioned earlier, parsing them requires a bit of *workspace memory*.

Incoming requests and cached responses are parsed in the client context, and use *client workspace memory*. When a cache miss takes place, and the response needs to be parsed from the origin server, we operate in the backend context. This will consume *backend workspace memory*.

There are certain limits in place that prevent *Varnish* from having to waste too much memory on request and response parsing and to avoid *DoS* attacks.

Here's an overview of runtime parameters that limit the length and size of requests and responses:

- `http_max_hdr`: the maximum number of headers an HTTP request or response may contain. The default value is *64*.
- `http_req_hdr_len`: the maximum size of an individual request header. By default this is *8 KB*.
- `http_req_size`: the maximum total size of the HTTP request headers. This defaults to *32 KB*.
- `http_resp_hdr_len`: the maximum size of an individual response header. By default this is *8 KB*.
- `http_resp_size`: the maximum total size of the HTTP response headers. This defaults to *32 KB*.

When requests or responses exceed these limits, the transaction will fail.

HTTP request limit examples

Here's some example logging output when the `http_max_hdr` threshold is exceeded:

```
*   << Request  >> 5
-   Begin          req 4 rxreq
-   Timestamp      Start: 1611051232.286266 0.000000 0.000000
-   Timestamp      Req: 1611051232.286266 0.000000 0.000000
-   BogoHeader     Too many headers: foo:bar
-   HttpGarbage    "GET%00"
-   RespProtocol   HTTP/1.1
-   RespStatus     400
-   RespReason     Bad Request
-   ReqAcct        519 0 519 28 0 28
-   End
```

As you can see, an *HTTP 400* status code is returned when this happens.

Here's an example where an individual request header exceeds the `http_req_hdr_len` limit:

```
*   << Request  >> 98314
-   Begin          req 98313 rxreq
-   Timestamp      Start: 1611051653.320914 0.000000 0.000000
-   Timestamp      Req: 1611051653.320914 0.000000 0.000000
-   BogoHeader     Header too long: test:YnEJyVqxTMgn7aX
-   HttpGarbage    "HEAD%00"
-   RespProtocol   HTTP/1.1
-   RespStatus     400
-   RespReason     Bad Request
-   ReqAcct        10081 0 10081 28 0 28
-   End
```

When the total request size exceeds `http_req_size`, the following output can be found in your *VSL*:

```
*   << Session  >> 32793
-   Begin          sess 0 HTTP/1
-   SessOpen       172.21.0.1 60576 http 172.21.0.3 80
1611052643.429084 30
-   SessClose      RX_OVERFLOW 0.001
-   End
```

HTTP response limit examples

When the origin server returns too many headers and exceeds the `http_max_hdr` limit, this doesn't result in an *HTTP 400* status, but in an actual *HTTP 503.*

You might see the following output appear in your *VSL*:

```
-- BogoHeader     Too many headers:foo: bar
-- HttpGarbage    "HTTP/1.1%00"
-- BerespStatus   503
-- BerespReason   Service Unavailable
-- FetchError     http format error
```

And when this happens, the `MAIN.losthdr` counter will also increase.

When the `http_resp_hdr_len` limit is exceeded, you will see the following output end up in *VSL*:

```
--  BogoHeader     Header too long: Last-Modified: Tue,
--  HttpGarbage    "HTTP/1.1%00"
--  BerespStatus   503
--  BerespReason   Service Unavailable
--  FetchError     http format error
```

And finally, when the `http_resp_size` limit is exceeded, the following *VSL* line may serve as an indicator:

```
--  FetchError     overflow
```

Make sure you have enough workspace memory

Remember, HTTP header processing, both for requests and responses, is done using *workspace memory*. If you decide to increase some of the HTTP header limits in `varnishd`, there's no guarantee that *Varnish* will work flawlessly.

The HTTP limits are there for a reason, and the defaults have been chosen pragmatically. When for example a *client workspace overflow* occurs, you'll see the following occur in your *VSL*:

```
-   Error          workspace_client overflow
-   RespProtocol   HTTP/1.1
-   RespStatus     500
-   RespReason     Internal Server Error
```

Interestingly, the status code is *HTTP 500* and not *HTTP 503*. This makes sense because the backend didn't fail; it's actually *Varnish* that failed.

In the *backend context*, *Varnish* is more likely to drop response headers that would cause a *backend workspace overflow* rather than fail the transaction.

When this happens, you'll see `LostHeader` tags appear in your *VSL* output:

```
--  LostHeader     foo:
--  LostHeader     bar:
```

If you really need to increase some of the HTTP limits, please ensure the workspace size is updated accordingly.

Limiting I/O with tmpfs

Varnish uses the `/var/lib/varnish` folder quite extensively. It's the place where the *VSL circular buffer* is located. It's also the place where the compiled *VCL* files are stored as `.so` files.

To avoid that *VSL* causes too many I/O operations, we can mount `/var/lib/varnish` as a `tmpfs` volume. This means `/var/lib/varnish` is actually stored in memory on a *RAM disk*.

These are the commands you need to make it happen:

```
echo "tmpfs /var/lib/varnish tmpfs defaults,noatime 0 0" | sudo tee
-a /etc/fstab
sudo mount /var/lib/varnish
```

You'll agree that this is a no-brainer.

7.10.6 Other settings

There are some other random settings that aren't deserving of their own subsection but can be useful nonetheless.

Listen depth

`listen_depth` is one of those settings. It refers to the number of unacknowledged pending TCP connections that are allowed in *Varnish*. On really busy systems, setting the queue high enough will yield better results.

But there is a fine line between better results and increased server load. The default value for `listen_depth` is set to *1024* connections.

However, this value is ignored if the operating system's `somaxconn` value is lower. Please verify the contents of `/proc/sys/net/core/somaxconn` to be sure.

If your operating system's value is too low, you can tune it via `sysctl -w net.core.somaxconn=1024`.

Nuke limit

This section doesn't apply to MSE, which will still use LRU eviction but relies on different mechanisms to keep things in check.

When you have a lot of inserts and your cache is full, the nuking mechanism will remove the *least recently used objects* to free the required space.

The `MAIN.n_lru_nuked` counter indicates that *LRU nuking* is taking place.

When a lot of small objects are stored in the cache, and a large objects needs to be inserted, the nuking mechanism may need to remove multiple objects before having enough space to store the new object.

There is an upper limit as to how many objects can be removed before the eviction is aborted. This is defined by the `nuke_limit` parameter. The standard value is set to 50.

If more than 50 object removals are required to free up space, *Varnish* will abort the transaction and return an *HTTP 503* error. The `MAIN.n_lru_limited` counter will count the number of times the nuke limit was reached.

Unfortunately, *Varnish Cache* has a limitation where a task can request *LRU nuking*, but where another competing task will steal its space. This might also be a reason why the `nuke_limit` threshold is reached.

Short-lived

The `shortlived` parameter will enforce the threshold for *short-lived objects*. This means that objects with a *TTL* lower than the value of `shortlived` will not be stored in the regular *caching stevedore*. Instead these objects will be stored in *transient storage*.

You may remember that *transient storage* is unbounded by default. This can result in your server going out of memory when the transient objects rapidly increase. By default the `shortlived` threshold is set to *ten seconds*.

Objects where the *full TTL*, which also implies the *grace and keep values*, is lower than *ten seconds* will go into the *transient storage*.

Logging CLI traffic in syslog

By default *CLI commands* are logged to *syslog* via `syslog(LOG_INFO)`. On systems that rely a lot on the CLI, this may result in a lot of noise in the logs, but also in degraded performance.

If you don't care that CLI commands are not logged, just set `syslog_cli_traffic` to `off`. It's always a tradeoff unfortunately.

7.11 The Varnish CLI

The *Varnish CLI* is a command-line interface offered by `varnishd` to perform a series of management tasks. The *Varnish CLI* has its own protocol that is accessible via TCP/IP.

Varnish also ships with a `varnishadm` program that facilitates CLI access.

Tasks that can be performed via the CLI are:

- Listing backends
- Administratively setting the backend health
- Ban objects from the cache
- Display the ban list
- Show and clear panics
- Show, set, and reset runtime parameters
- Display process id information
- Perform liveliness checks to `varnishd`
- Return status information
- Starting and stopping the *Varnish child process*
- Manage VCL configurations

This is what the *Varnish CLI* looks like when called using the `varnishadm` program:

```
$ varnishadm
200
-----------------------------
Varnish Cache CLI 1.0
-----------------------------
Linux,5.4.39-linuxkit,x86_64,-junix,-sdefault,-sdefault,-hcritbit
varnish-6.0.7 revision 525d371e3ea0e0c38edd7baf0f80dc226560f26e

Type ‘help’ for command list.
Type ‘quit’ to close CLI session.
```

CLI commands can be run inside the `varnishadm` shell, but they can also be appended as arguments to the `varnishadm` program.

7.11.1 Backend commands

The `backend.list` command lists all available backends, and also provides health information, as you can see in the example below:

```
varnish> backend.list
200
Backend name                   Admin      Probe                Last
updated
boot.default                   probe      Healthy             7/8
Tue, 05 Jan 2021 12:34:08 GMT
boot.static-eu                 probe      Healthy (no probe)   Tue,
05 Jan 2021 12:34:08 GMT
boot.static-us                 probe      Healthy             7/8
Tue, 05 Jan 2021 12:34:08 GMT
```

If you add a `-p` option to the command, you'll start seeing more detailed information on the health probe:

```
varnish> backend.list -p
200
Backend name                   Admin      Probe                Last
updated
boot.default                   probe      Healthy             8/8
  Current states  good:  8 threshold:  3 window:  8
  Average response time of good probes: 0.004571
  Oldest ================================================== Newest
  ----------------------------------------------------4444444444
Good IPv4
  ----------------------------------------------------XXXXXXXXXX
Good Xmit
  ----------------------------------------------------RRRRRRRRRR
Good Recv
  --------------------------------------------------HHHHHHHHHHHH
Happy
 Tue, 05 Jan 2021 12:34:08 GMT
boot.static-eu                 probe      Healthy (no probe)   Tue,
05 Jan 2021 12:34:08 GMT
boot.static-us                 probe      Healthy             8/8
  Current states  good:  8 threshold:  3 window:  8
  Average response time of good probes: 0.004552
  Oldest ================================================== Newest
  -----------------------------------------------------44444444444
Good IPv4
  -----------------------------------------------------XXXXXXXXXXX
Good Xmit
  -----------------------------------------------------RRRRRRRRRRR
Good Recv
  ---------------------------------------------------HHHHHHHHHHHHH
Happy
 Tue, 05 Jan 2021 12:34:08 GMT
```

If you have a large number of backends, listing detailed information for all of them can become unmanageable. You can narrow down the scope by supplying a backend pattern to the `backend.list` command.

The following example only lists backends that start with `static`. Evidently, the `boot.static-eu` and the `boot.static-us` backends will appear:

```
varnish> backend.list -p static*
200
Backend name                   Admin      Probe                Last
updated
boot.static-eu                 probe      Healthy (no probe)   Tue,
05 Jan 2021 12:34:08 GMT
boot.static-us                 probe      Healthy             8/8
  Current states  good:  8 threshold:  3 window:  8
  Average response time of good probes: 0.004164
  Oldest ================================================== Newest
  -----------------------44444444444444444444444444444444444444444
Good IPv4
  -----------------------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Good Xmit
  -----------------------RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
Good Recv
  ---------------------HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
Happy
 Tue, 05 Jan 2021 12:34:08 GMT
```

We can use the `backend.set_health` command to override the health of one or more backends, based on a backend pattern.

For example, when downtime is expected for a group of backends, it makes sense to explicitly set them to unhealthy beforehand. If the backends are governed by a director, they will automatically be taken out of the directors rotation, which is a more graceful approach to a planned outage.

Here's an example where we set both `static-eu` and `static-us` to an unhealthy state:

```
varnish> backend.set_health static-* sick
200
```

When we list the backends, we can see that the `Admin` field no longer contains the `probe` value, but the `sick` value:

```
varnish> backend.list
200
Backend name                   Admin      Probe                Last
updated
boot.default                   probe      Healthy             8/8
Tue, 05 Jan 2021 12:34:08 GMT
boot.static-eu                 sick       Healthy (no probe)   Tue,
05 Jan 2021 12:42:36 GMT
boot.static-us                 sick       Healthy             8/8
Tue, 05 Jan 2021 12:42:36 GMT
```

Let's go ahead and set the health of the two backends to `auto`. This will undo our previous `backends.set_health` command, setting their health back to the value as listed under the `Probe` field.

```
varnish> backend.set_health static-* auto
200
```

You can also force a backend to be considered healthy, as illustrated in the example below:

```
varnish> backend.set_health static-* healthy
200
```

7.11.2 Banning

We already talked about *banning* via the CLI in *chapter 6*. We'd like to refer to that part of the book for more details.

As a quick reminder, here's an example of a ban issued via the CLI:

```
varnish> ban "obj.http.Content-Type ~ ^image/"
200
```

And here's a *ban list* example:

```
varnish> ban.list
200
Present bans:
1609850980.159475     0 -   obj.http.Content-Type ~ ^image/
```

7.11.3 Parameter management

The CLI has various commands to set the value of a parameter, list its value, and reset it to the default value.

Displaying parameters

The `param.show` command lists the value of the configurable runtime parameters inside *Varnish*.

When running this command without additional options or arguments, you get a list of parameters with their value.

Here's an extract because the full list is a bit too long:

```
varnish> param.show
200
accept_filter                 -
acceptor_sleep_decay          0.9 (default)
acceptor_sleep_incr           0.000 [seconds] (default)
acceptor_sleep_max            0.050 [seconds] (default)
auto_restart                  on [bool] (default)
backend_idle_timeout          60.000 [seconds] (default)
backend_local_error_holddown  10.000 [seconds] (default)
backend_remote_error_holddown 0.250 [seconds] (default)
ban_cutoff                    0 [bans] (default)
ban_dups                      on [bool] (default)
ban_lurker_age                60.000 [seconds] (default)
ban_lurker_batch              1000 (default)
ban_lurker_holdoff            0.010 [seconds] (default)
ban_lurker_sleep              0.010 [seconds] (default)
between_bytes_timeout         60.000 [seconds] (default)
...
```

You can also get this output with a lot more context and meaning. Just add the `-l` option, as you can see in the extract below:

```
varnish> param.show -l
200
acceptor_sleep_decay
        Value is: 0.9 (default)
        Minimum is: 0
        Maximum is: 1

        If we run out of resources, such as file descriptors or work-
er
        threads, the acceptor will sleep between accepts.
        This parameter (multiplicatively) reduce the sleep duration
for
        each successful accept. (ie: 0.9 = reduce by 10%)

        NB: We do not know yet if it is a good idea to change this
        parameter, or if the default value is even sensible. Caution
is
        advised, and feedback is most welcome.

acceptor_sleep_incr
        Value is: 0.000 [seconds] (default)
        Minimum is: 0.000
        Maximum is: 1.000

        If we run out of resources, such as file descriptors or work-
er
        threads, the acceptor will sleep between accepts.
        This parameter control how much longer we sleep, each time we
        fail to accept a new connection.

        NB: We do not know yet if it is a good idea to change this
        parameter, or if the default value is even sensible. Caution
is
        advised, and feedback is most welcome.
...
```

It is also possible to only list the parameters where the value was changed. To achieve this, just use the `param.show changed` command.

Here's some example output:

```
varnish> param.show changed
200
feature                       +http2
shortlived                    5.000 [seconds]
thread_pool_max               7500 [threads]
```

In this case, we added the `http2` feature flag, modified the timing for short-lived objects to *five seconds*, and set the maximum number of threads in a thread pool to *7500 threads*.

You can also get the value of an individual parameter, as shown in the example below:

```
varnish> param.show shortlived
200
shortlived
        Value is: 5.000 [seconds]
        Default is: 10.000
        Minimum is: 0.000

        Objects created with (ttl+grace+keep) shorter than this are
        always put in transient storage.
```

It is even possible to list the output in *JSON format* by adding a `-j` option. Here's an example where we display information about the `default_ttl` parameter in *JSON format*:

```
varnish> param.show -j default_ttl
200
[ 2, ["param.show", "-j", "default_ttl"], 1609857571.607,
  {
    "name": "default_ttl",
    "implemented": true,
    "value": 120.000,
    "units": "seconds",
    "default": "120.000",
    "minimum": "0.000",
    "description": "The TTL assigned to objects if neither the back-
end nor the VCL code assigns one.",
    "flags": [
      "obj_sticky"
]
}
]
```

Setting parameter values

The `param.set` command assigns a new value to a parameter, which is quite convenient because it doesn't require restarting the `varnishd` process.

The downside of setting parameters via the CLI is that the changes are not persisted. As soon as `varnishd` gets restarted, the values that were assigned by `-p` are used, and other values are reset to their default value.

The `param.set` command is great for temporary changes, or for changes where a `varnishd` restart is not desirable. If you want a parameter change to be persisted, just add the appropriate `-p` option to your `varnishd` startup script.

Here's an example of a parameter change where we set the `default_ttl` parameter to one minute:

```
varnish> param.set default_ttl 60
200
```

But if you need to undo the change and want to reset the parameter to its default value, just run `param.reset`:

```
varnish> param.reset default_ttl
200
```

7.11.4 VCL management

Another important feature of the *Varnish CLI* is the *VCL management* capability. This is especially useful from a *VCL deployment* point of view.

You can load multiple *VCL configurations*, set an active one, and even assign labels so that inactive *VCL code* can be conditionally loaded into your main *VCL file*.

VCL inspection

Commands like `vcl.list` and `vcl.show` can be used to list the available *VCL configurations* and to show the corresponding *VCL code*.

When you start *Varnish*, this is probably the output you'll get:

```
varnish> vcl.list
200
active      auto/warm          0 boot
```

We have a single active *VCL configuration*, which is called `boot`. If we want to see the *VCL code* for this configuration, we run `vcl.show boot`, as illustrated below:

```
varnish> vcl.show boot
200
vcl 4.1;

backend default {
    .host="localhost";
    .port="8080";
}
```

Loading VCL

If you want multiple *VCL configurations* to be loaded, you can add one or more configurations by running the `vcl.load` command.

As you can see, the command requires a configuration name and a path to the *VCL file*:

```
varnish> vcl.load server1 /etc/varnish/server1.vcl
200
VCL compiled.
```

The `vcl.load` command will compile the code and bail out if an error was encountered. This is also an interesting way to check whether your *VCL* is syntactically correct.

If you don't want to depend on a *VCL file*, you can directly inject a *quoted VCL string* via `vcl.inline`. The quoting sometimes gets a bit tricky, but here's a very simple example:

```
varnish> vcl.inline default << EOF
varnish> vcl 4.1;
varnish>
varnish> backend be {
varnish>     .host="localhost";
varnish>     .port="8080";
varnish> }
varnish> EOF
200
VCL compiled.
```

If we want the previously loaded *VCL configuration* to be active, just run the following command:

```
varnish> vcl.use server1
200
VCL 'server1' now active
```

Don't forget that inactive *VCL configurations* still consume resources. If you no longer need older *VCL configurations*, it is advisable to remove them using the `vcl.discard` command, as the next example shows:

```
varnish> vcl.discard boot
200
```

VCL labels

VCL labels have two purposes.

They behave like symbolic links to actual *VCL configurations* and can be used to switch from one VCL configuration to another.

Here's an example where we assign the `my_label` label to the `my_configuration` VCL configuration:

```
varnish> vcl.label my_label my_configuration
200
```

At this point `my_label` will be listed as such and can be used with other VCL commands:

```
varnish> vcl.list
200
active      auto/warm          0 my_configuration
available  label/warm          0 my_label -> my_configuration
varnish> vcl.use my_label
200
VCL 'my_label' now active
varnish> vcl.list
200
available   auto/warm          0 my_configuration
active     label/warm          0 my_label -> my_configuration
```

Multiple labels can point to the same *VCL configuration*, but a label cannot point to another label. This can be useful to maintain abstract *VCL configurations*. You could imagine having one label called `production` and another called `maintenance` to easily switch from one to the other during an outage, without needing to know in detail which exact *VCL configuration* should be used for either scenario. You can update and roll back the underlying *VCLs* independently and separate VCL management from VCL selection.

But the second purpose of *VCL labels* is probably the most useful. The *active VCL* is allowed to switch to a different VCL in the `vcl_recv` subroutine. This allows you to maintain multiple concurrent *VCL configurations* independently, which can greatly help *virtual hosting* when multiple applications need very different cache policies.

Imagine a situation where multiple *VCL configurations* are loaded, one for each web application it is caching:

```
varnish> vcl.load www_1 www.vcl
200
VCL compiled.
varnish> vcl.load api_1 api.vcl
200
VCL compiled.
```

As you can see, on top of the default configuration, we also have the `www_1` and `api_1` configurations.

We can label these configurations, as illustrated below:

```
varnish> vcl.label www www_1
200
varnish> vcl.label api api_1
200
varnish> vcl.label www_example_com www_1
200
varnish> vcl.label api_example_com api_1
200
```

- The `www_1` config has labels `www` and `www_example_com`
- The `api_1` config has labels `api` and `api_example_com`

From within our main *VCL file*, we'll load various *labeled VCL configurations* based on the *host header* of the request.

Here's the main *VCL file* that loads the labels:

```
vcl 4.1;
import std;

backend default none;

sub vcl_recv {
    if (req.http.Host == "www.example.com") {
        return(vcl(www));
    } elseif (req.http.Host == "api.example.com") {
        return(vcl(api));
    } else {
        return(synth(404));
    }
}
```

- If a request is received containing the `Host: www.example.com` request header, the `www` label is used
- If a request is received containing the `Host: api.example.com` request header, the `api` label is used

Each labeled *VCL configuration* has its own logic, and its own backends. This allows for multi-tenancy to some extent.

The `vcl.list` command then shows the labels, and how they are used:

```
varnish> vcl.list
200
available   auto/warm          0 www_1 (2 labels)
available   auto/warm          0 api_1 (2 labels)
available  label/warm          0 www -> www_1 (1 return(vcl))
available  label/warm          0 api -> api_1 (1 return(vcl))
active      auto/warm          0 default
available  label/warm          0 www_example_com -> www_1
available  label/warm          0 api_example_com -> api_1
```

The configurations themselves have a reference counter that keeps track of how many times they were used by a label. The labels point to the configuration they are associated with. And if any of these labels are used within a `return(vcl())` statement, this is also mentioned. In this example the regular web application lives alongside an HTTP API, and they both have different cache policies and can be updated independently:

```
varnish> vcl.load www_2 www.vcl
200
VCL compiled.
varnish> vcl.label www www_2
200
varnish> vcl.label www_example_com www2
200
varnish> vcl.list
200
available   auto/cold          0 www_1
available   auto/warm          0 api_1 (2 labels)
available  label/warm          0 www -> www_2 (1 return(vcl))
available  label/warm          0 api -> api_1 (1 return(vcl))
active      auto/warm          0 default
available  label/warm          0 www_example_com -> www_2
available  label/warm          0 api_example_com -> api_1
available   auto/warm          0 www_2 (2 labels)
```

Rolling back is only a matter of labeling `www_1` again if the `www_2` update wasn't correct, without disturbing the API.

VCL temperature

VCL configurations consume resources, even when they are not active. If you deploy a new version of your *VCL* and keep the previous versions, the allocated resources for these *VCL files* will not be released immediately.

Varnish has a built-in system to cool down *VCL configurations* when they are no longer in use. Resources that were reserved by these *VCLs* are eventually released.

When a new *VCL configuration* is deployed, it becomes warm. This is done automatically, but the `vcl.state` command allows you to override the *VCL temperature*.

The *VCL temperature* can be set to one of the following values:

- `auto`
- `warm`
- `cold`

The `vcl.list` command lists the various configurations, but also includes the temperature, and how it was set:

```
varnish> vcl.list
200

active      auto/warm          0 default
available   auto/cold          0 test
```

In this case, the `default` configuration, which is the active one, is in the `auto/warm` state. The `test` configuration is available, but no longer in use. It has become *cold*.

If we want to force the temperature, we can use the `vcl.state` command to warm up or cool down the configuration:

```
varnish> vcl.state test warm
200
```

In this example we explicitly set the state to `warm`, which is also reflected in the *VCL list*:

```
varnish> vcl.list
200

active      auto/warm          0 default
available   warm/warm          0 test
```

We can also set it back to `auto`:

```
varnish> vcl.state test auto
200
varnish> vcl.list
200

active      auto/warm          0 default
available   auto/warm          0 test
```

7.11.5 Configuring remote CLI access

If you're planning on connecting to the *Varnish CLI* remotely, it makes sense to tune the remote CLI access runtime parameters.

The `-T` runtime parameter sets the listening address and port for the CLI. The `-S` runtime parameter is used to define the location of the secret file.

This secret file contains the secret key that is required to gain access to the CLI.

You probably want to see these two parameters in action, so here's an example:

```
varnishd -a :80 -T localhost:6082 -S /etc/varnish/secret -f /etc/var-
nish/default.vcl
```

This is a pretty basic `varnishd` configuration where the CLI is only accessible locally using *port 6082*. The authentication protocol uses the contents of the `/etc/varnish/secret` file.

The `varnishadm` command is capable of connecting to a remote CLI. The `-T` and `-S` options are also available for `varnishadm`.

Here's an example of a remote ban using `varnishadm`:

```
varnishadm -S /etc/varnish/secret -T varnish.example.com:6082 ban
"obj.http.x-url == /info"
```

7.11.6 The CLI protocol

The *Varnish CLI* has its own CLI protocol, which is largely abstracted away when using `varnishadm`. But if you want to integrate the *Varnish CLI* into your own application, you need to understand the protocol.

From your application, you'll connect to the host and port that were configured using the `-T` parameter. In the example below this is `localhost:6082` because the application happens to run on the same machine as *Varnish*.

For security reasons, access will be restricted based on the secret key that was set using the `-S` parameter.

Here's an example where we connect to the CLI via `telnet`. The assumption is that `/etc/varnish/secret` contains `my-big-secret` as its value.

Here's the output:

```
$ telnet localhost 6082
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
107 59
lgsefmatosfviyytnnrbwrvwngdkhkrn

Authentication required.
```

- `107` is the status code that means *authentication is required*.
- The next line contains `lgsefmatosfviyytnnrbwrvwngdkhkrn`, which is *the challenge*.

Based on this *challenge* and the *secret* from `/etc/varnish/secret`, the *authentication string* is composed.

You start by creating a string that contains the following parts:

- The *challenge*
- A newline (`\0x0a`)
- The *secret*
- The *challenge* again
- A newline (`\0x0a`)

If we use `my-big-secret\n` as the secret key, this would be our string:

```
lgsefmatosfviyytnnrbwrvwngdkhkrn
my-big-secret
lgsefmatosfviyytnnrbwrvwngdkhkrn
```

This string then needs be hashed via the *SHA256* hashing algorithm, and the resulting digest should be returned in *lowercase hex*.

The end result would be the following *authentication string*:

```
b931c0995b200b83645a4e4e9bbb9061b2c80c2aaa878920d8b2da8612756f5c
```

The response to the challenge in the *Varnish CLI* would be `auth <authentication string>`. In our case, this is what happens:

```
auth b931c0995b200b83645a4e4e9bbb9061b2c80c2aaa878920d8b2da8612756f5c
200 277
-----------------------------
Varnish Cache CLI 1.0
-----------------------------
Linux,5.4.39-linuxkit,x86_64,-junix,-sdefault,-sdefault,-hcritbit
varnish-6.0.7 revision 525d371e3ea0e0c38edd7baf0f80dc226560f26e

Type 'help' for command list.
Type 'quit' to close CLI session.
```

Because status code `200` was returned, we know the authentication procedure was successful. We get the banner from the *Varnish CLI*, and we can start executing CLI commands.

For your convenience we have created a small *bash script* that will create the *authentication string* for you:

```
#!/bin/sh

set -e

exec </etc/varnish/secret

if [ $# = 0 ]; then
    echo "Challenge not set, exiting" >&2
    exit 1
fi

(
    printf '%s\n' "$1"
    cat
    printf '%s\n' "$1"
) |
sha256sum |
awk '{print $1}'
```

The script checks whether or not `/etc/varnish/secret` exists and whether or not the challenge was passed as a *command line argument*.

The *authentication string* is created and sent to `sha256sum` to create the *SHA256 digest*. Finally we send the output to the `awk` program to fetch the first part, which is the final *authentication string*.

Here's how you would invoke the script: `/auth.sh <challenge>`. And here's the script in action:

```
$ ./auth.sh lgsefmatosfviyytnnrbwrvwngdkhkrn
b931c0995b200b83645a4e4e9bbb9061b2c80c2aaa878920d8b2da8612756f5c
```

And as expected, `b931c0995b200b83645a4e4e9bbb9061b2c80c2aaa878920d8b2da8612756f5c` is the output you can use to respond to the *authentication challenge* that was imposed by the *Varnish CLI*.

7.11.7 The CLI command file

We already mentioned the fact that changes through the *Varnish CLI* are not persisted. This means that a `varnishd` restart will undo your changes.

One of the solutions we suggested, especially for `param.set` commands, was to also add the customizations in your `varnishd` startup script via `-p` runtime parameters.

This can work for parameter tuning, but for other commands it doesn't. Take for example the `vcl.label` command: if you depend on *VCL labels*, a `varnishd` restart can result in effective downtime.

To avoid any drama, `varnishd` has a `-I` option that points to a *CLI command file*. This contains CLI commands that are executed when `varnishd` is launched.

This way, you can ensure your *VCL labels* are correctly set, and the corresponding *VCL files* are loaded when you start or restart *Varnish*.

```
vcl.load s1 /etc/varnish/server1.vcl
vcl.load s2 /etc/varnish/server2.vcl
vcl.label server1 s1
vcl.label server2 s2
```

If these commands are stored inside `/etc/varnish/clifile`, the following example loads this file:

```
varnishd -a :80 -f /etc/varnish/default.vcl -I /etc/varnish/clifile
```

If any of the commands fail, `varnishd` will not properly start. Commands that are prefixed with `-` will not abort `varnishd` startup upon failure.

7.11.8 Quoting pitfalls

Using quoted or multi-line strings in the CLI can lead to unexpected behavior.

Expansion

CLI commands take a set number of arguments. If one of the arguments happens to be a multi-word string, you'll need to use quotes. However, if you run these commands outside of the CLI shell and inside the shell of your operating system, double expansion takes place.

The quoting examples hinge on the fact that we want to override the `cc_command` runtime parameter. In reality you'll rarely change the value of this parameter. We selected this example because it's one of the few parameters that takes a string argument.

Imagine that we want to set the value of `cc_command` to `my alternate cc command`.

You might set the parameter as follows:

```
varnish> param.set cc_command my alternate cc command
105
Too many parameters
```

But as you can see, only `my` is used as the value. The other parts of the string are considered extra arguments. This is a problem, and the solution is to add quotes, as illustrated in the following example:

```
varnish> param.set cc_command "my alternate cc command"
200

Change will take effect when VCL script is reloaded
```

If we make this example into a one-liner outside of the `varnishadm` CLI scope and execute it in the operating system's shell, you'll get the following result:

```
$ varnishadm param.set cc_command "my alternate cc command"
Too many parameters

Command failed with error code 105
```

The string is expanded twice, resulting in the error we saw previously. To work around this, we can add extra quotes.

```
$ varnishadm param.set cc_command '"my alternate cc command"'

Change will take effect when VCL script is reloaded
```

If you want to parse an environment variable into a `varnishadm` CLI command, more quoting magic takes place:

```
$ TEST="Varnish"
$ varnishadm param.set cc_command '"'$TEST'"'

Change will take effect when VCL script is reloaded
```

The extra pair of single quotes is required, otherwise `$TEST` will be passed as an unparsed string.

Heredoc

If you want to pass multi-line content using the CLI, you may use *Heredoc* notation.

Here's an example where we use the `cat` program to output a string. This string happens to be a multi-line one, that is defined using *Heredoc* syntax:

```
$ cat <<EOF
Thijs
Feryn
EOF
Thijs
Feryn
```

Within the *Varnish CLI*, you cannot use `<<EOF` as the start of a multi-line string. Here's what you get when you do:

```
varnish> vcl.inline test <<EOF
106
Message from VCC-compiler:
VCL version declaration missing
Update your VCL to Version 4 syntax, and add
    vcl 4.1;
on the first line of the VCL files.
('<vcl.inline>' Line 1 Pos 1)
<<EOF
##---
```

```
Running VCC-compiler failed, exited with 2
VCL compilation failed
```

To make *Heredoc*-style input work, you need to add a space between `<<` and `EOF`, as illustrated below:

```
varnish> vcl.inline test << EOF
varnish> vcl 4.1;
varnish> backend default none;
varnish> sub vcl_recv {
varnish>     return(synth(200));
varnish> }
varnish> EOF
200
VCL compiled.
```

Remember: the *Varnish CLI* format for *Heredoc* text requires an extra space, but outside of the *CLI* scope this no longer applies. If you want this to work using a `varnishadm` one-liner, you need to quote the *Heredoc*.

Here's the previous example again, but inserted from outside of the *CLI* scope:

```
$ varnishadm vcl.inline test '<< EOF
vcl 4.1;
backend default none;
sub vcl_recv {
    return(synth(200));
}
EOF'
VCL compiled.
```

7.12 The Varnish Controller

Managing a cluster of *Varnish* servers, deploying *Varnish Configuration Language (VCL)*, tuning the right parameters, monitoring availability, and assigning roles to groups of servers. These are all operational responsibilities that can become quite a hassle when a lot of nodes are being managed.

There are plenty of tools out there that can be used for some of these tasks. However, the *Varnish Controller* is a specialized solution that integrates all these responsibilities.

The *Varnish Controller* is developed by *Varnish Software* and was the logical successor of the now deprecated *Varnish Administration Console (VAC)*.

> At the time of writing, the *Varnish Controller* is a brand-new product. The current features are primarily focused on *VCL deployment*. By the time you read this, the project may have gone through several iterations, be more feature-rich, and may look and behave a bit differently.

The *Varnish Controller* is a commercial product provided by *Varnish Software* and is not available in an open source incarnation.

7.12.1 Architecture

The *Varnish Controller* was designed to be extremely flexible: the different services of the project can be run separately and can be scaled horizontally.

Communication with the controller is done via a *RESTful API*. These API calls are received by the *API gateway*. And all the way at the end, there are *agents* running on the *Varnish* instances that leverage the *varnish management interface port* to perform management tasks.

For the services in between the *API gateway* and the *agents*, there is a lot of *inter-process communication* going on. Communication between the various services is done using the *Neural Autonomic Transport System (NATS)* messaging service.

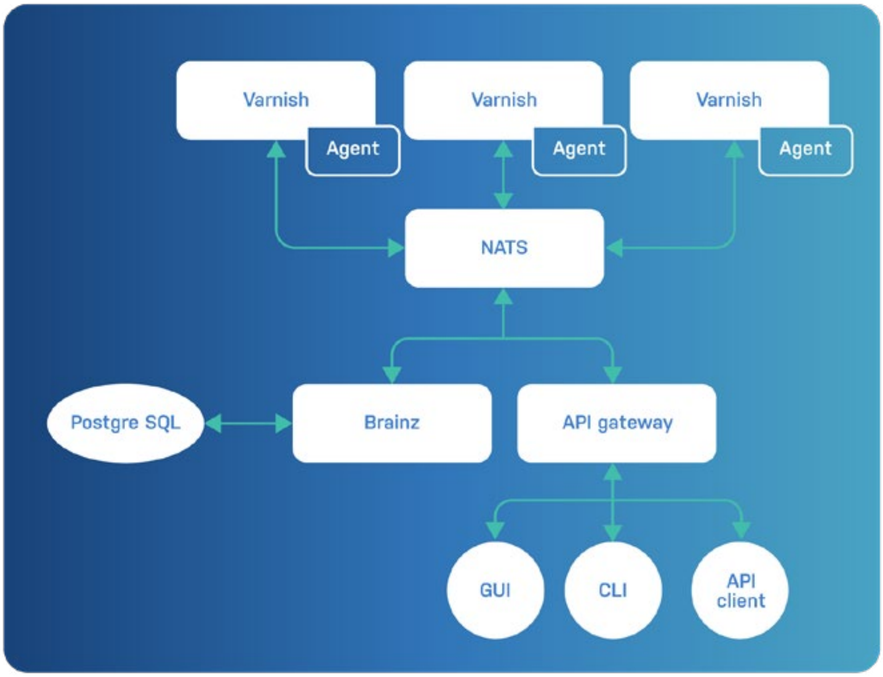
NATS is a lightweight, high-performance messaging service. The agents receive messages over *NATS*, which contain instructions, but the message queue is also used to report information.

The central service that manages all information and keeps track of the state is called the *Brainz*. The *Brainz* service reads from and writes to *NATS*, stores state in a *Post-*

greSQL database, and also communicates with the *API gateway* over *NATS*. Any decision-making in the system is handled by the *Brainz*, or one of the *Brainz*, if running more than one.

And in the end, the *API* can be consumed directly or via clients like the *command line interface (CLI)* or the *graphical user interface (GUI)*.

Here's an architecture diagram containing the various services and how they are interconnected:



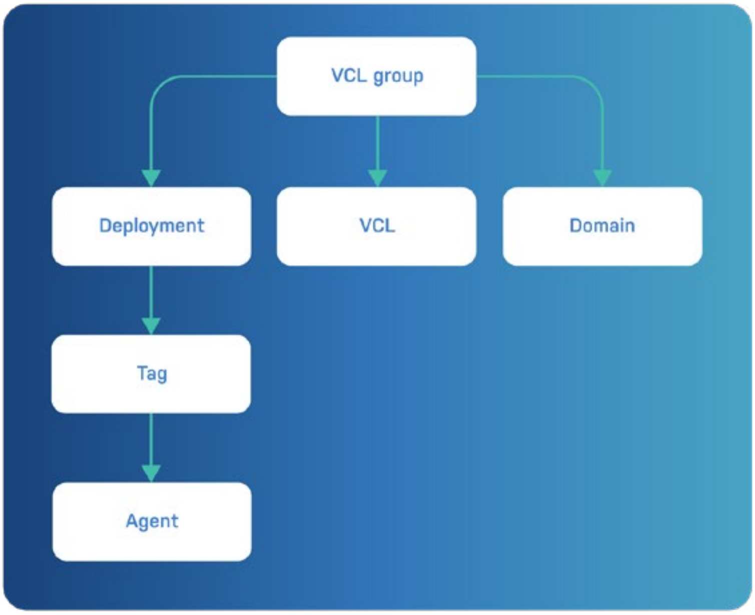
Varnish Controller architecture

In this diagram, the core of the *Varnish Controller* doesn't provide any redundancy. This was done for the sake of simplicity. In real life, you'll have at least a cluster of *NATS* servers, a cluster of *PostgreSQL databases*, and multiple instances of *Brainz* and the *API gateway*.

7.11.1 Core concepts

Within the *Varnish Controller*, there are a couple of core concepts present. The concepts dictate how *VCL code* is going to end up on *Varnish* servers.

You can see these concepts, and how they are connected, in the following diagram:



Varnish Controller core concepts

Let's talk about the individual concepts. You'll notice that they are nearly all related to *VCL* in some way.

Domain

When we talk about multi-tenancy, it's not only about managing multiple organizations in one central place. It's also about deploying *VCL* for multiple domains.

The controller is designed in such a way that a single *Varnish* server can host *VCL configurations* for multiple domains without interference.

Behind the scenes, the *Varnish Controller* generates a routing *VCL* using *VCL labels* to load the right *VCL configuration* for the right domain. The *Varnish Controller* needs exclusive access to the *Varnish* instance so any existing *VCL code* is unloaded from servers that are registered with the *Varnish Controller*.

It is also possible not to use domains, in which case the deployment is considered *root deployment* where you deploy one *VCL* per server, or one *VCL* to many servers. Each server in the *Varnish Controller* can either be a multi-tenant or a *root deploy* node.

VCL

At this point, *VCL* shouldn't be a hard to concept to grasp. The controller manages a collection of *VCL* files that can be deployed to *Varnish* servers via the *agents* that run on them.

The *VCL code* could either be loaded for a specific domain or for the entire server it runs on.

VCL configurations can either be the *main VCL file* of the configuration, or one or more *include files* that can be included in the main configuration. The *main VCL file* is the file that gets used in *Varnish*, and any include statement will then load the accompanying files as the *VCL* is parsed by *Varnish*.

Deployment

A deployment is an entity that links *VCL configurations* to eligible agents. This is done using *tags*: agents can have one or more tags and can be linked with multiple tags as well.

But a deployment will not deploy the selected *VCL configurations* to all matching agents. The deployment can be limited with a *minimum* and a *maximum* parameter.

If for example the minimum is set to *two agents* and the maximum to *three agents*, the deployment will at least try to find *two available agents* to consider the deployment successful. If *three agents* are available, the *VCL* will be deployed to all three agents.

Creating this range facilitates autoscaling. As the capacity of the cluster grows, more agents can register themselves with the controller, which results in automatic deployment if more agents register with the same tags. These can be both static tags or assigned tags.

VCL group

The *VCL configurations* can be grouped into *VCL groups*. The *VCL group* is what ties it all together. As you may have noticed, *VCL groups* is at the top of the diagram of *core concepts*.

When defining a *VCL group*, a *VCL configuration* needs to be selected and one or more deployments as well.

If the *VCL group* is to be used only for requests matching specific domains, the domains will also be linked in the group. If not, the group considers this a dedicated server where the *VCL* acts as the *root VCL*.

Agent

The *agent* is a small service that runs on the *Varnish* servers that are managed by the *Varnish Controller*.

As mentioned, the agent leverages the *Varnish management interface* to perform management tasks. If the agent is configured in *read-only mode*, it will only use the management interface to retrieve information.

Otherwise, the agent will also use the management interface to deploy *VCL* to the *Varnish server*.

7.11.1 Setup

If you have a *Varnish Enterprise* subscription, you can use packages to install the various components. The individual components can be installed on the same machine, on separate servers, or in any combination.

Whether you're using *Debian*, *Ubuntu*, *Red Hat*, or *CentOS*, the packages can be installed using the package manager of your choice.

The following packages are to be installed for the controller to work:

- `varnish-controller-nats`
- `varnish-controller-brainz`
- `varnish-controller-api-gw`

Of course, you'll need a *PostgreSQL* installation as well.

On the individual *Varnish* servers that are to be managed by the controller, the `varnish-controller-agent` package needs to be installed.

If you want to interact with the *Varnish Controller* using the *CLI*, the `varnish-controller-cli` package can be installed.

If you want to manage your *Varnish* cluster using a graphical user interface, you can install the `varnish-controller-ui` package.

Both the *CLI* and the *GUI* leverage the *Varnish Controller API*, whereas the other components interact via the *NATS* message queue.

7.11.1 Authentication & authorization

To keep access to the *Varnish Controller* as secure as possible, you can configure authorized access.

Either *basic authentication* can be used to gain access to the controller, but we also provide *KeyCloak* integration. *KeyCloak* is an open source identity and access management solution that supports multiple identity providers. You can use it to facilitate *single sign-on*, *user federation* and *social login* using *OAuth2* and *OpenID Connect*.

You can create separate organizations, each with its own users. Users can be organization administrators or regular users. For each user, fine-grained permissions can be set. Even organizations can have different *KeyCloak* integrations or *basic authentication*.

7.11.1 The API

The primary interface of the *Varnish Controller* is the *RESTful API*. Any other type of client will also use the *API* to get things done.

The first thing you need to do when you want to access or change resources via the *API* is to log in.

The `/api/v1/auth/login` endpoint does exactly that. Here's an example:

```
$ curl -X POST -utest:test http://localhost:8080/api/v1/auth/login
{"accessExpire":1611232848,"accessToken":"eyJhbG...",
"refreshExpire":1611235848,"refreshToken":"eyJhbG..."}
```

Via the username `test` and password `test` we manage to create an *access token* and a *refresh token*. The *access token* can be used in subsequent *API calls* in the form of a *Bearer Authentication Token*.

> This example uses `http://localhost:8080` and the *API gateway* endpoint. It also uses very simplistic credentials that are presented using *basic authentication*. This is fine for testing, but in production, it will look quite different.

Here's an easy way to extract the `accessToken` from the *JSON output* via the `jq` program. The extracted value is stored on disk in the `access.token` file.

```
$ curl -s -X POST -utest:test http://localhost:8080/api/v1/auth/login
| jq -r ".accessToken" > access.token
```

Instead of having to paste the lengthy token in your `curl` command, you can directly load it from the file, as illustrated in this example:

```
$ curl -v -H "Authorization: Bearer $(cat accesstoken)" http://local-
host:8080/api/v1/agents | jq
[
  {
    "agentVersion": "1.0.1 - b9d5f3a892b8d8faef5e2986b8be18822f-
4d64ef",
    "created": "2021-01-21T11:50:34.838827Z",
    "id": 1,
    "lastHeartbeat": "2021-01-21T12:38:22.555717Z",
    "lastStateChange": "2021-01-21T11:50:34.841335Z",
    "name": "agent1",
    "state": 1,
    "tags": [
      {
        "created": "2021-01-21T11:50:34.855042Z",
        "id": 1,
        "name": "prod",
        "static": true,
        "updated": "2021-01-21T12:38:15.563396Z"
      }
    ],
    "updated": "2021-01-21T12:38:22.556023Z",
    "varnishHost": "192.168.99.102",
    "varnishPort": 6081,
    "varnishState": "running",
    "varnishVersion": "plus-6.0.6r10 revision 048baeea9cfe2cd133e-
5115da7e1efa26d7901eb"
  }
]
```

we'll keep using `jq`, because it makes the *JSON* output a lot more readable.

Here's a final example where we'll upload a *VCL file* to the controller:

```
$ curl -X POST -H "Authorization: Bearer $(cat accesstoken)" "http://
localhost:8002/api/v1/vcls" -d '{ "id": 2, "name": "synthok",
"source": "vcl 4.1;\n \n backend default none;\n \n sub vcl_recv {\n
return(synth(200,\"ok\"));\n }"}' | jq
{
  "contentType": "text/plain; charset=utf-8",
  "created": "2021-01-21T13:17:50.3050258Z",
  "encoded": false,
  "fileType": 1,
  "id": 5,
  "name": "synthok",
  "sha": "\"1736e032c6dd3d45d8aa81782b6c7131b481437ae86c90a-
c909260176a524cb3\"",
  "source": "vcl 4.1;\n \n backend default none;\n \n sub vcl_recv
{\n     return(synth(200,\"ok\"));\n }",
  "updated": "2021-01-21T13:17:50.3050258Z"
}
```

Full *Swagger API*-based documentation is available via `http://localhost:8080/docs/index.html`. Please change the hostname and the port of this *URL* to the endpoint of your *Varnish Controller API gateway*.

7.11.1 The CLI

The *API* is nice, it works great, it is clean, but it's still an *API* and not that user-friendly.

For people who are comfortable working on the command line, using the *Varnish Controller CLI* is a significant improvement in terms of user experience.

If you have the `varnish-controller-cli` package installed, you can easily interact using the `vcli` program.

Logging in is done via `vcli login`. You can pass the *API endpoint*, your username, password and organization:

```
$ vcli login http://localhost:8002 -u test -p test -o myorg
Configuration saved to: ~/.vcli.yml
Login successful.
```

Please note that the *login URL* shouldn't have an ending `/`.

`vcli` keeps track of the state and stores the access and refresh tokens in `~/.vcli.yml`. The next time you login, you can use the `vcli login` command and the *refresh token* will ensure you have access again.

If you're logging in using admin credentials, there's no need to mention the organization during the login procedure.

Listing agents is as simple as running `vcli agent list`

```
$ vcli agent list
+----+--------+---------+----------------+-----------------+---------
+
| ID |  Name  |  State  |  Varnish Host  | Varnish Version |   Tags
|
+----+--------+---------+----------------+-----------------+---------
+
|  1 | agent1 | Running | 192.168.99.102 | plus-6.0.6r10   | (1)prod
|
+----+--------+---------+----------------+-----------------+---------
+
```

Here are the available `vcli` commands:

- `account`
- `agent`
- `apilog`
- `deployment`
- `domains`
- `file`
- `help`
- `idp`
- `login`
- `logout`
- `organization`
- `permission`
- `session`
- `tags`
- `util`
- `vcl`
- `vclgroup`
- `version`

When you use the `help` subcommand, you'll get a lot more information about other subcommands and their capabilities. Here's an example of what `vcli vclgroups` is capable of:

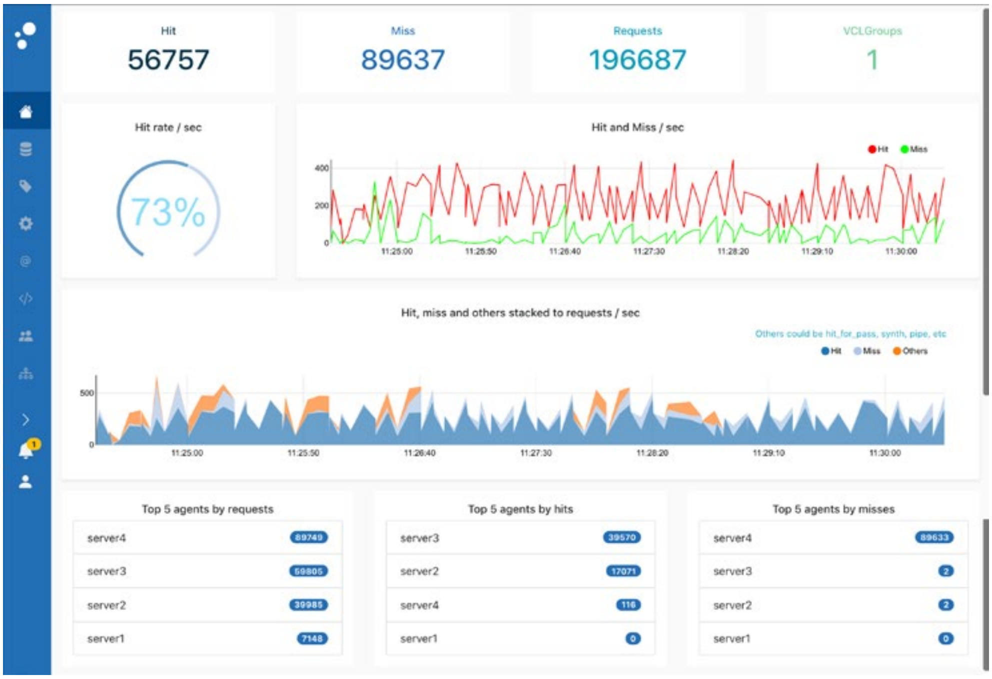
```
$ vcli vclgroup help
Handle VCLGroup such as listing, add, delete or update.

Examples:
  vcli vclgroups inspect 1
  vcli vclgroups list
  vcli vclgroups list -f name="*MyVCL*"
  vcli vclgroups add MyVCLGroupName --vcl 1 --dep 1 --root
  vcli vclgroups update 1 --name newName --root=false --dep 1,2
  vcli vclgroups delete 1
  vcli vclgroups stage 1
  vcli vclgroups promote 1
  vcli vclgroups delstage 1
  vcli vclgroups stats 1

Usage:
  vcli vclgroup [command]

Aliases:
  vclgroup, vg, vgs, vclgroups

Available Commands:
  add          Add new VCLGroup
  delete       Remove a VCLGroup
  delstage     Delete a staging for a VCLGroup
  deploy       Deploy a VCLGroup
  inspect      Inspect a VCLGroup's details
  list         List VCLGroups
  promote      Promote a staged VCLGroup from staging to production
  reload       Reload a VCLGroup on servers
  stage        Stage a VCLGroup
  stats        Show varnish statistics for VCLGroups
  undeploy     Undeploy a VCLGroup
  update       Update a vclGroup
```

7.11.1 The GUI

In terms of speed of execution for simple tasks, it's tough to beat the efficiency of the *CLI*. However, a graphical user interface makes the controller a lot more user-friendly.

Whereas the *CLI* and *API* present the core concepts as individual resources, the *GUI* will group these concepts into an intuitive interface. Some of the complexities are nicely hidden.

Here are a couple of screenshots:



Top 5 agents by requests

server4	89749
server3	59805
server2	39985
server1	7148

Top 5 agents by hits

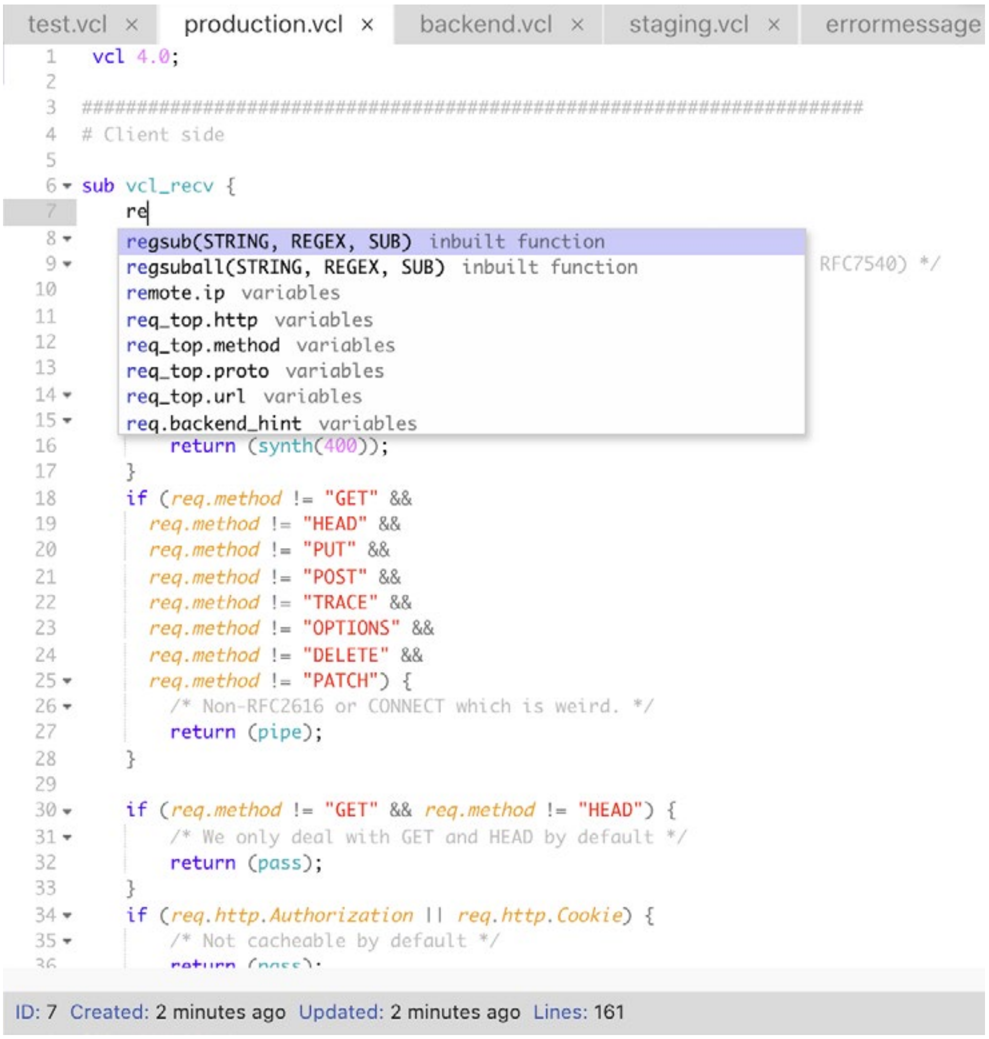
server3	39570
server2	17071
server4	116
server1	0

Top 5 agents by misses

server4	89633
server3	2
server2	2
server1	0

Varnish Controller dashboard

The first screenshot features the dashboard. It contains an overview of various counters, graphs, and lists. As this project evolves, more useful metrics will be integrated into the dashboard.



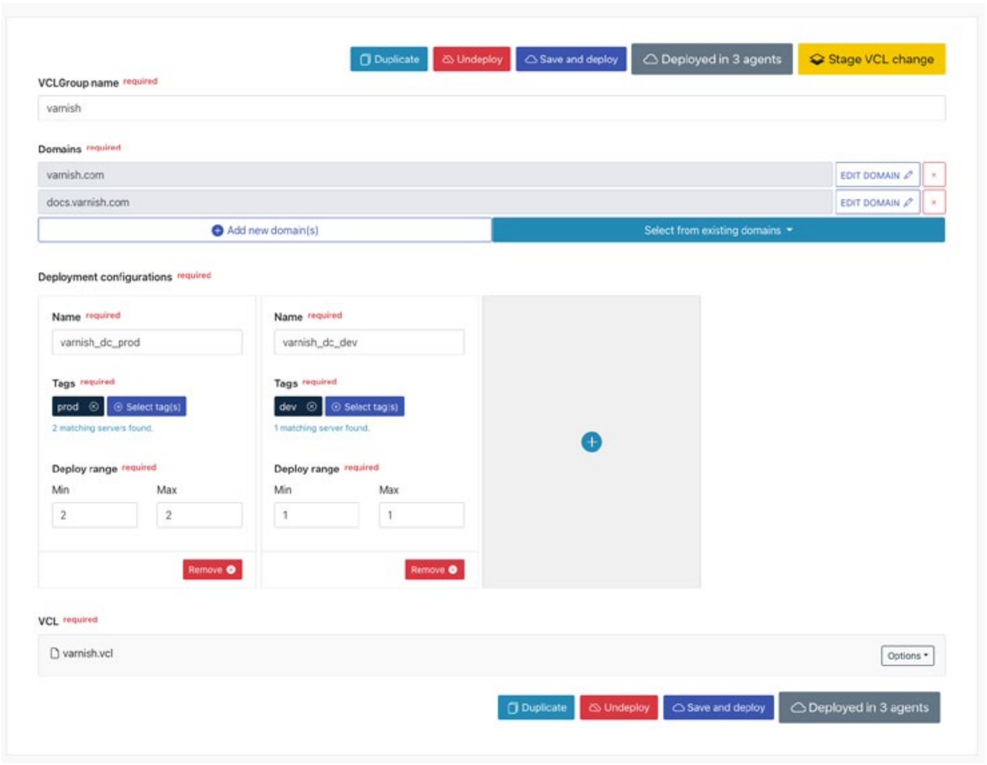
test.vcl × | production.vcl × | backend.vcl × | staging.vcl × | errormessage

```
vcl 4.0;

#######################################################################
# Client side

sub vcl_recv {
    re
    regsub(STRING, REGEX, SUB)  inbuilt function
    regsuball(STRING, REGEX, SUB)  inbuilt function
    remote.ip  variables
    req_top.http  variables
    req_top.method  variables
    req_top.proto  variables
    req_top.url  variables
    req.backend_hint  variables
                                                   RFC7540) */
        return (synth(400));
    }
    if (req.method != "GET" &&
      req.method != "HEAD" &&
      req.method != "PUT" &&
      req.method != "POST" &&
      req.method != "TRACE" &&
      req.method != "OPTIONS" &&
      req.method != "DELETE" &&
      req.method != "PATCH") {
        /* Non-RFC2616 or CONNECT which is weird. */
        return (pipe);
    }

    if (req.method != "GET" && req.method != "HEAD") {
        /* We only deal with GET and HEAD by default */
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie) {
        /* Not cacheable by default */
        return (pass);
```

ID: 7 Created: 2 minutes ago Updated: 2 minutes ago Lines: 161

Varnish Controller VCL editor

The next screenshot features the built-in *VCL editor*. Not only can you upload *VCL files*, but you can also create new ones. The editor has *syntax highlighting* and *code completion*, which offers an excellent experience for composing *VCL code*.



VCLGroup name required

varnish

Domains required

varnish.com

docs.varnish.com

Deployment configurations required

Name required: varnish_dc_prod	Name required: varnish_dc_dev
Tags required: prod	Tags required: dev
2 matching servers found.	1 matching server found.
Deploy range required: Min 2, Max 2	Deploy range required: Min 1, Max 1

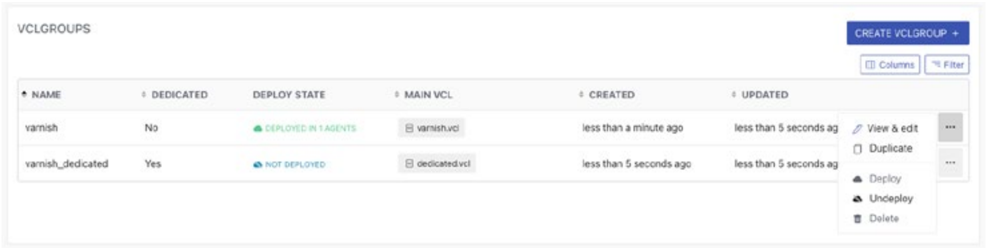
VCL required

varnish.vcl

Varnish Controller VCL group creation

The following screenshots show how various configuration concepts are nicely integrated into a single workflow. This clearly shows how *VCL groups*, *deployments*, *VCL files*, and *domains* are connected.

The end result is a deployable *VCL group*, and the last screenshot shows the overview of available configurations:



NAME	DEDICATED	DEPLOY STATE	MAIN VCL	CREATED	UPDATED
varnish	No	DEPLOYED IN 1 AGENTS	varnish.vcl	less than a minute ago	less than 5 seconds ag
varnish_dedicated	Yes	NOT DEPLOYED	dedicated.vcl	less than 5 seconds ago	less than 5 seconds ag

Varnish Controller VCL group creation

Please note that by the time you read this book, or by the time you're considering using the *Varnish Controller*, the *GUI* might look a bit different. Despite stable releases, this product is still in development. There will still be a lot of feature updates, but also cosmetic updates that will impact the look and feel of the *GUI*. Follow https://docs.varnish-software.com/varnish-controller/changelog/ for *Varnish Controller* changelogs.

7.13 Summary

Thanks for sticking around because that was quite the trip. This chapter is by far the longest one, but in our opinion also the most exciting one.

It goes to show that there's a lot more to *Varnish* than writing some *VCL* and putting *Varnish* in front of your web server.

Deploying *Varnish* at scale and using it to build a *content delivery platform* results in extra concerns and requirements that go beyond a single `varnishd` instance with *256MB* of object caching.

Hopefully this chapter inspired you to use the various tools that the *Varnish* project offers.

There's no denying that *Varnish*'s logging and monitoring tools offer unprecedented levels of insight compared to regular web servers.

And although *Varnish Cache* is used by millions of websites at incredible scale, this chapter has shown where *Varnish Enterprise* really shines. In terms of security, high availability, custom statistics, large-scale persistent storage and cluster management, *Varnish Enterprise* provides the goods.

Don't get me wrong, although I work at *Varnish Software*, and although I use *Varnish Enterprise* on a daily basis, this is not a commercial pitch. This book is all about features and capabilities, and hopefully you are convinced about the technical capabilities of both the project and the product.

In my experience, *Varnish* usually ends up being the responsibility of the *ops team*. That's why this *Varnish for operations* chapter deserves the amount of effort, the level of detail, the diverse topics, and the page count that we've put into it.

Get ready for *chapter 8* where we will take *VCL* to the next level. *Varnish* is more than a *take-it-or-leave-it* cache, and even in situations where application state prevents us from caching, there are ways to implement stateful logic *on the edge* and still cache the content.

Chapter 8: Decision-making on the edge

Welcome to *chapter 8* where we will focus on decision-making on the edge.

Varnish is more *than a take-it-or-leave-it cache*. Most *Varnish* implementations focus on caching as much as possible, using *VCL* to explicitly include content, and exclude content that is not cacheable. The *take-it* part refers to cacheable content, the *leave-it* part to non-cacheable content.

Despite good hit rates and acceptable performance at a large scale, the non-cacheable content can still be a weak spot.

The reasons why certain content cannot be cached can sometimes be quite trivial:

- Authentication gets in the way
- Simple logic based on the value of a cookie

By leveraging *VMODs* and bypassing *built-in VCL logic*, it is entirely possible to cache otherwise uncacheable content.

Varnish can act as an authentication gateway; *Varnish* can serve personalized content without creating too many cache variations; *Varnish* can even interact with third-party systems to feed the cache with external data.

Let's write some *VCL* to do more on the edge and to increase our hit rate.

8.1 Dealing with state

Caching stateful content is very challenging, and out-of-the-box *Varnish* will not cache this kind of content.

Here's the *built-in VCL* extract that proves this:

```
if (req.http.Authorization || req.http.Cookie) {
    /* Not cacheable by default */
    return (pass);
}
```

If the client presents an `Authorization` header or a `Cookie` header, the request is stateful, and the corresponding response can therefore not be served from cache.

Even if the client request doesn't introduce state, it is possible that a backend response can force a state change. Here's a modified and simplified version of how *Varnish* deals with this at the backend level in its *built-in VCL*:

```
sub vcl_backend_response {
    if(beresp.http.Set-Cookie) {
        set beresp.ttl = 120s;
        set beresp.uncacheable = true;
    }
    return(deliver);
}
```

You will probably remember this from *chapters 3 and 4*, where we talked about the *built-in VCL* in a lot of detail: if the origin introduces a `Set-Cookie` header, it implies a state change. Therefore *Varnish* will deem the response uncacheable.

The reality is that most websites use cookies and that the *built-in VCL* is not sufficiently equipped for real-world applications, as its caching policy is overly cautious. In *chapter 4* there is a section called *Making changes* where we showed you how to write *VCL* that gets values from cookies, and how you can create cache validations using cookies.

Let's crank it up a notch, and use `Cookie` and `Authorization` header values to deliver cacheable personalized content. These headers are mainly used as identifiers. The actual content comes from an external service: either an *API* or a *database*.

The idea is that the origin server no longer serves stateful output for select content. Instead the origin will serve a template where the placeholder is replaced by personalized data coming from an external system. All of this happens *on the edge*, which means we're really *caching the uncacheable*.

The rest of this chapter features concepts and examples that relate to this idea.

8.2 Body access

Before we can start personalizing the caching experience, we need to cover some fundamentals.

An important one is understanding how to access the *body* of an HTTP request or response.

A *request body* may contain a field or a parameter that we will use to identify the user. This request body information can also be used to create a cache variation.

We can also inspect and rewrite the *response body*. This means we can modify the output *on the edge* without having to access the origin application.

8.2.1 Request body access

Let's start with the *request body*. The request body doesn't usually occur in a `GET` method. Although it is theoretically possible, *Varnish* just strips it off before sending the request to the origin.

This means the *request body* is used for requests that use a `POST`, `PUT`, `PATCH`, or maybe even a `DELETE` method.

Unfortunately there is no `req.body` variable in *VCL*, and `bereq.body` can only be unset.

Accessing the request body starts by calling `std.cache_req_body()`. This function call is required to ensure that the request body is read from the client and stored in memory. Otherwise, the request body could only be read once and could not be accessed in other parts of the *VCL*.

The function takes an argument that defines the size of a cacheable request body.

The function argument is a `byte` type, as illustrated below:

```
vcl 4.1;

import std;

sub vcl_recv {
    if(std.cache_req_body(1KB)) {
        std.log("Request body accessible");
    } else {
        std.log("Request body not accessible");
    }
}
```

This means the request body is only accessible if the size is smaller than *1 KB*.

Once `std.cache_req_body()` has been called, there is a variety of *VMODs* we can use to leverage this information.

Storing the request body in memory by calling `std.cache_req_body` also makes sense when you *restart* or *retry* a transaction. Because the request body needs to be sent to the origin multiple times, we cannot afford losing this information. That's why we cache it.

vmod_bodyaccess

The easiest way to access the request body is by using `vmod_bodyaccess`. It is an open source *VMOD* that is part of the *Varnish Software VMOD collection*. See *chapter 5* to refresh your memory.

`vmod_bodyaccess` has a pretty limited *API* and is able to hash the request body, calculate the length of the request body, search for strings in the body, and log the request body to the *VSL*.

Let's store the request body in the hash first. Here's how you do this:

```
vcl 4.1;

import std;
import bodyaccess;

sub vcl_recv {
    std.cache_req_body(10KB);
    set req.http.x-method = req.method;
    return(hash);
}

sub vcl_hash {
    bodyaccess.hash_req_body();
}

sub vcl_backend_fetch {
    set bereq.method = bereq.http.x-method;
}
```

Because *built-in VCL* doesn't allow `POST` calls to be served from cache, we must override this behavior in `vcl_recv`. It starts with explicitly calling `return(hash)` to bypass standard behavior.

Another trick we must do is store the original *request method* in a custom request header. If we try to cache a `POST` call, *Varnish* strips the request body and turns the request into a `GET` request.

That's why we reset the request method inside `vcl_backend_fetch`.

It's also pretty obvious that we're only caching request bodies that are at most *10 KB* in size. The actual caching happens by adding `bodyaccess.hash_req_body()` to `vcl_hash`.

When you run the following command, you'll get the output specifically for the `key=value` *post data*:

```
curl -XPOST -d "key=value" localhost
```

And when you change the *post data* to `key=otherValue`, the output will be different:

```
curl -XPOST -d "key=otherValue" localhost
```

Just look at the `Age` header. It will tell how long it has been in cache. Changes in *post data* will result in *cache variations*, which will result in different values for the `Age` header.

The `bodyaccess.rematch_req_body()` function allows us to inspect the request body, making it possible to reject unwanted requests and make request body caching conditional. Here's an example:

```
vcl 4.1;

import std;
import bodyaccess;

sub vcl_recv {
    std.cache_req_body(10KB);
    if(bodyaccess.rematch_req_body("key=[^=]+") == 1) {
        set req.http.x-method = req.method;
        return(hash);
    }
}

sub vcl_backend_fetch {
    if(bereq.http.x-method) {
        set bereq.method = bereq.http.x-method;
    }
}

sub vcl_hash {
    bodyaccess.hash_req_body();
}
```

In this example, we only cache requests where the request body contains a field named `key`. If it is set to `key=value`, then the request body is cached.

The following `curl` call is cacheable:

```
curl -XPOST -d "key=value" localhost
```

And because the following `curl` call doesn't match the `key=[^=]+` regular expression, it is not cacheable:

```
curl -XPOST -d "key" localhost
```

If you're interested in what the actual length of the request body was, you can use the `bodyaccess.len_req_body()` function.

And if you want the request body to be visible in `varnishlog`, you can leverage `bodyaccess.log_req_body(STRING prefix = "", INT length = 200)`, which takes a *prefix* and a *max line length* argument, so the body could be split up across multiple lines.

xbody

A more feature-rich alternative to `vmod_bodyaccess` is *xbody*. It's part of *Varnish Enterprise*, and it's a very useful tool in your *edge-computing* toolbox.

`vmod_xbody` is capable of caching the *request body*, just like `vmod_bodyaccess`, but it can also access the response body. And more importantly, the module is also capable of modifying request and response bodies.

Here's the `vmod_xbody` equivalent of request body caching:

```
vcl 4.1;

import xbody;
import std;
import blob;

sub vcl_recv {
    std.cache_req_body(10KB);
    if(xbody.get_req_body() ~ "key=[^=]+") {
        set req.http.x-method = req.method;
        set req.http.x-hash = blob.encode(encoding=BASE64,blob=xbody.
get_req_body_hash(md5));
        return(hash);
    }
}

sub vcl_backend_fetch {
    if(bereq.http.x-method) {
        set bereq.method = bereq.http.x-method;
    }
}

sub vcl_hash {
    hash_data(req.http.x-hash);
}
```

You may have noticed that we execute the `xbody.get_req_body_hash()` function from within `vcl_recv`. That's because this function is only accessible from that subroutine. The return type of this function is a `BLOB`, so we need `vmod_blob` to turn it into a string.

In the end, we can use the `x-hash` request header to transport the request body hash to the `vcl_hash` subroutine.

Because `xbody.get_req_body()` returns a string, we can make our cache variations more efficient. The `vmod_bodyaccess` example used the *entire request body* as a cache variation. But if we use `regsub()` we can choose the exact *part of the request body* we want to vary on.

Here's an example where we only create variations on the value of the `key` field:

```
vcl 4.1;

import xbody;
import std;
```

```
sub vcl_recv {
    std.cache_req_body(10KB);
    if(xbody.get_req_body() ~ "(^|.+&)key=([^\=\&]+)(&.+|$)") {
        set req.http.x-method = req.method;
        set req.http.x-hash = regsub(xbody.get_req_body(),"(^|.+&)
key=([^\=\&]+)(&.+|$)","\2");
        return(hash);
    }
}

sub vcl_backend_fetch {
    if(bereq.http.x-method) {
        set bereq.method = bereq.http.x-method;
    }
}

sub vcl_hash {
    hash_data(req.http.x-hash);
}
```

You've probably spotted that the regular expression we use to match the key is more complicated. That's true, but because it's also more intelligent: `key=value` is a typical pattern we try to match. But without the extra regex logic, `mykey=value` would also match.

The following `curl` call would create a variation for the term `value`:

```
curl -XPOST -d "key=value" localhost
```

The following `curl` call, which has a different request body, would also hit that same variation:

```
curl -XPOST -d "foo=bar&key=value" localhost
```

By being more deliberate about the way we create *request body variations*, we can significantly increase our hit rate.

And because our regular expression is more secure, the following `curl` call would miss that variation and result in a cache miss:

```
curl -XPOST -d "mykey=value" localhost
```

json.parse_req_body()

vmod_json is a *Varnish Enterprise* module that can parse *JSON data* and can return individual *JSON fields*. From a *request body* point of view, the json.parse_req_body() function is of particular interest to us.

Let's revisit the earlier examples, and try to cache POST requests using the function.

```
vcl 4.1;
import json;
import std;

sub vcl_recv {
    std.cache_req_body(10KB);
    json.parse_req_body();
    if (json.is_valid() && json.is_object() && json.get("key")) {
        set req.http.x-method = req.method;
        return(hash);
    }
}

sub vcl_backend_fetch {
    if(bereq.http.x-method) {
        set bereq.method = bereq.http.x-method;
    }
}

sub vcl_hash {
    hash_data(json.get("key"));
}
```

The json.parse_req_body() function in this example will parse the request body as *JSON* and store the result in a new JSON context. Via json.get() we can fetch individual values at a later stage.

However, via json.is_valid() we can check whether or not the valid *JSON* was parsed. Via json.is_object() we can check whether or not the *JSON* data is an object. And finally, we check whether or not the key property is found inside the *JSON* object by using json.get("key").

If all of these conditions apply, we can look the object up in cache, even if the request is a POST request. If not, the *built-in VCL* will handle it from there.

And just like in the previous example, we only create variations on specific fields. In this case the value of the key property.

Here's a `curl` call where the payload matches the criteria, which results in this `POST` call being cached:

```
curl -XPOST -d "{ \"key\": \"value\"}" localhost
```

Even though the following example has a different *JSON request payload*, it will also match the initial variation because the `key` property exists:

```
curl -XPOST -d "{ \"key\": \"value\", \"foo\": \"bar\" }" localhost
```

8.2.2 Response body access

Analyzing and changing the *response body* is where it gets really exciting.

In very basic terms, you can change the response body by setting the `beresp.body` and `resp.body` variables.

Unfortunately their usage is very restricted. `resp.body` can only be set in `vcl_synth`, as illustrated below:

```
vcl 4.1;

sub vcl_recv {
    return(synth(200));
}

sub vcl_synth {
    set  resp.body = "Welcome";
    return(deliver);
}
```

And `beresp.body` can only be set in `vcl_backend_error`:

```
vcl 4.1;

backend default none;

sub vcl_backend_error {
    set  beresp.body = "Welcome";
    return(deliver);
}
```

Alternatively, the `synthetic()` function can be used to achieve the same, and depending on the subroutine it is used in, either `beresp.body` or `resp.body` will be set.

xbody revisited

Remember *xbody*? As mentioned this *VMOD* can also inspect and modify the response body.

We promise to present really good examples where `vmod_xbody` and `vmod_edgestash` are combined. But that's for later; first let us show you some really basic examples:

Imagine the following obnoxiously hypothetical *response body*:

```
Hello world
```

The following *VCL example* will replace `world` with the *IP address* of the client:

```
vcl 4.1;
import xbody;

sub vcl_backend_response {
    xbody.regsub("Hello \w+","Hello " + client.ip);
}
```

The end result on our local computer would be:

```
Hello 192.168.16.1
```

We can also use the `xbody.capture()` function to capture values that we can retrieve using `xbody.get()` and `xbody.get_all()` afterwards:

```
vcl 4.1;

import xbody;
import std;

sub vcl_backend_response {
    xbody.capture("name","Hello (\w+)","\1");
}

sub vcl_deliver {
    std.log("Name: " + xbody.get("name"));
}
```

Although the *response body* remains untouched when we use xbody.capture, the captured value will be logged in *VSL*:

```
$ varnishlog -g raw -I VCL_Log:Name
     32770 VCL_Log        c Name: world
```

Trust us: we'll show you a more exciting example once we've introduced you to vmod_edgestash.

Edgestash

Speaking of which, *Edgestash* is one of our favorite *Varnish Enterprise* features, which is available through vmod_edgestash.

You've probably heard of *Mustache*, a simple handlebars-based templating language that originated in the *JavaScript world*. It has tons of implementations on other languages and is somewhat of an industry standard in terms of templating.

Edgestash is a module that processes *Mustache handlebars*. Basically, you have *Mustache on the edge*, or *Edgestash*, if you will.

The idea is that placeholders like {{variable}} are put into your templates. The business logic of your application is responsible for parsing the values into those placeholders.

An origin application can emit a placeholder for potentially *non-cacheable, personalized* content, and have *Varnish* cache the otherwise uncacheable page and populate it with its required value.

This value may be identified by a session cookie or authentication credentials. The basic business logic that identifies the user and collects the stateful information can be offloaded to *Varnish*. *Edgestash* will be responsible for assembling the bits and pieces and parsing it into a single *HTTP response body*.

Imagine that your origin application returns the following output:

```
Hello {{name}}
```

The {{name}} placeholder could then be replaced by the *client IP address* using the following *VCL code*:

```
vcl 4.1;

import edgestash;

sub vcl_backend_response {
    if(beresp.http.edgestash) {
        edgestash.parse_response();
    }
}

sub vcl_deliver {
    if (edgestash.is_edgestash()) {
        edgestash.add_json({"
        {
            "name":""} + client.ip + {""
        }
        "});
        edgestash.execute();
    }
}
```

At first, it doesn't seem more interesting than the `xbody.regsub()` example. However, not only does *Edgestash* support the full *Mustache syntax*, the parsing happens at delivery time in `vcl_deliver` instead of at cache-insertion time in `vcl_backend_response`.

This means values could be injected on-the-fly. It's also important to note that *JSON* is the basis of the parsing.

It's also important to note that the previous *VCL* example only processes *Mustache handlebars* when the response contains an `edgestash` response header. This avoids wasting CPU cycles on non-Mustache content.

This is the parsed *JSON* that is processed by *Edgestash*:

```
{
    "name":"192.168.16.1"
}
```

And this is the final output:

```
Hello 192.168.16.1
```

JSON endpoint

Manually composing a *JSON* string in `edgestash.add_json()` can be clunky at times. A very elegant way to inject *JSON* is by using the `edgestash.add_json_url()`.

This function takes an *HTTP endpoint* as its first argument and expects the response to be *JSON* output. *RESTful APIs* are excellent candidates for these endpoints.

This allows you to split cacheable responses and stateful content into separate endpoints.

Here's an example:

```
vcl 4.1;

import edgestash;

sub vcl_backend_response {
    if(bereq.url == "/api") {
        edgestash.index_json();
    } elseif(beresp.http.edgestash) {
        edgestash.parse_response();
    }
}

sub vcl_deliver {
    if (edgestash.is_edgestash()) {
        edgestash.add_json_url("/api");
        edgestash.execute();
    }
}
```

As long as the `/api` endpoint produces a *JSON* object that has a `name` property, the value can be parsed into the placeholder.

If the *JSON* endpoint is located on another domain, you can use the second argument to specify the hostname. This could end up being `edgestash.add_json_url("/api","api.example.com")`.

The `edgestash.index_json()` function inside `vcl_backend_response` will index the *JSON* for faster processing when `edgestash.execute()` is called.

Advanced Mustache templating

The *Mustache templating language* does more than replace placeholders with actual values.

It can perform loops; it has conditionals; there are variables and expression, and basic arithmetic.

Imagine the following *JSON* output, which represents a shopping cart:

```
[
  {
    "id": 1,
    "name": "Watch",
    "price": 25,
    "amount": 2
  },
  {
    "id": 2,
    "name": "Shoes",
    "price": 80,
    "amount": 1
  }
]
```

This is stateful data that depends on a *session cookie*. The `curl` call that is required to retrieve the *JSON* could be the following:

```
curl -s -H"Cookie: PHPSESSID=9755a8b773f76bffeda28f746ac3957e" local-
host/session
```

As you can see the `Cookie: PHPSESSID=9755a8b773f76bffeda28f746ac3957e` header is set to identify the user.

```
[
  {
    "id": 1,
    "name": "Watch",
    "price": 25,
    "amount": 2
  },
  {
    "id": 2,
    "name": "Shoes",
    "price": 80,
    "amount": 1
  }
]
```

The goal is to turn this *JSON* data into the following *HTML* code:

```
<ul>
    <li>Watch: 2 x 25 EUR = 50 EUR</li>
    <li>Shoes: 1 x 80 EUR = 80 EUR</li>
</ul>
```

This means we have to find a way to list the product name for each item in the cart, but also the price and the product quantity.

The following *Mustache syntax* would be required to do the job:

```
<ul>
{{#.}}
    <li>{{name}}: {{amount}} x {{price}} EUR = {{amount * price}}
EUR</li>
{{/}}
</ul>
```

The `{{#.}}...{{/.}}` expression can be used to iterate over a *JSON array*. The `{{amount * price}}` expression does a multiplication.

Whereas `{{#.}}{{/.}}` was used in the previous example to iterate through an array, `{{#name}}{{/name}}` could be used to check whether or not the `name` property exists.

Here's some example *JSON*:

```
{
    "name": "Thijs"
}
```

And here's the conditional:

```
{{#name}}Welcome {{name}}{{/name}}
{{^name}}Welcome guest{{/name}}
```

Under normal circumstances `Welcome Thijs` would be returned. If for some reason the `name` property is not in the *JSON* output, or the *JSON* endpoint is not accessible, `Welcome guest` would be returned.

8.2.3 An e-commerce example

In this subsection, we'll show you an example where we can combine *xbody* and *Edgestash* to cache personalized data.

The use case is an *e-commerce platform*. In this case it's written in *PHP* and uses the *Symfony* framework. There is a shopping cart that shows the number of items in the cart.

Sessions

The shopping cart is stored by the framework's session handler in the `/session` folder on disk. The *session id* could, for example, be `9755a8b773f76bffeda28f746ac3957e`.

The corresponding session file would be `/sessions/sess_9755a8b773f76bffeda28f746ac3957e`, and the cookie that tracks this session would be `Cookie: PHPSESSID=9755a8b773f76bffeda28f746ac3957e`.

Inside `sess_9755a8b773f76bffeda28f746ac3957e` you could find the following session data:

```
_sf2_attributes|a:2:{s:4:"cart";a:1:{i:1;i:9;}s:11:"itemsInCar-
t";i:9;}_sf2_meta|a:3:{s:1:"u";i:1611851104;s:1:"c";i:1611759335;s:1:
"l";s:1:"0";}
```

The session file is serialized using PHP's built-in serializer. It's not exactly *JSON*, but you can spot certain data structures. The number of items that this user has in the shopping cart is represented by `s:11:"itemsInCart";i:9;`. This means this user has nine items in the cart.

Cacheability

When you visit the *e-commerce platform*, this value is visible in the *HTML* source code:

```
<span id="items-in-cart">9</span>
```

When you don't have any items in the shopping cart, the *HTML* element remains empty, no session is initialized, and no cookie is set. This means the page is perfectly cacheable.

However, as soon as an element is stored in cache, the cookie is set:

```
Set-Cookie: PHPSESSID=4fde6819330b5d7d2166ae8fcab71a52; path=/; Http-
Only; SameSite=lax
```

The `Set-Cookie` will trigger a *hit-for-miss*, and subsequent requests that have the `Cookie` header will trigger a *pass*. This makes the platform uncacheable.

But even if we decide to cache despite the cookie, the shopping cart value will also be cached. This is not acceptable.

An alternative solution would be to create a *cache variation* per session id. Unfortunately, this will impact the hit rate.

The caching solution

The solution we're going to apply is a non-intrusive one that doesn't require any code changes.

First we're going to match `<span id="items-in-cart">9</span>` with `xbody.regsub` and inject *Edgestash* handlebars. This makes the page cacheable.

We're going to use `vmod_kvstore` to store the items in the cart inside *Varnish*. The *key-value store* has a value per session id. At delivery time `vmod_edgestash` will parse the value into the placeholder and display the right value per session. This happens without accessing the origin application.

The *key-value store* will be populated from the session file. Using `vmod_file` we can read the right session file, extract the `itemsInCart` value, and store it in the *key-value store*.

To avoid excessive file system access, we'll only read the session file when an item is added to or removed from the cart. This requires intercepting requests for `/add/to/cart/$id` and `/remove/from/cart/$id`.

The VCL code

Let's go over the *VCL code* for our non-intrusive caching solution:

```
vcl 4.1;
import edgestash;
import xbody;
import cookieplus;
import kvstore;
import file;
```

```
sub vcl_init {
    new cart = kvstore.init();
    new sessions = file.init("/sessions/");
}
```

As you can see, we need a number of *VMODs* to get the job done. In `vcl_init` we're initializing the *key-value store* as the `cart` object.

We're also configuring file system access by creating a `sessions` object that has access to the `/sessions` folder.

The next step involves checking for incoming requests:

```
sub vcl_recv {
    cookieplus.keep("PHPSESSID");
    cookieplus.write();
    if(req.url ~ "^/add/to/cart/[0-9]+$" || req.url ~ "^/remove/from/
cart/[0-9]+") {
        return(pass);
    }
    if(req.url == "/") {
        return(hash);
    }
}
```

We're making sure that only the `PHPSESSID` cookie is kept. Any other cookie is removed.

The next step involves intercepting requests to `/add/to/cart/$id` and `/remove/from/cart/$id`. When either of these is received we perform a `return(pass)` to make sure these pages aren't cached.

Time to see how *xbody* facilitates the use of *Edgestash* in `vcl_backend_response`:

```
sub vcl_backend_response {
    if(bereq.url == "/") {
        unset beresp.http.Cache-Control;
        set beresp.ttl = 3600s;
        xbody.regsub({"(<span id="items-in-cart" [^>]+>)(\w*)(</
span>)"},
            {"\1{{items-in-cart}}\3"});
        edgestash.parse_response();
    }
    if(bereq.url ~ "^/add/to/cart/[0-9]+$" || bereq.url ~ "^/remove/
from/cart/[0-9]+") {
        call refresh_cart;
    }
}
```

When we first receive the backend response from the origin server, we look for the *HTML element* that contains the items in cache.

`xbody.regsub()` will turn `<span id="items-in-cart">9</span>` into `<span id="items-in-cart">{{items-in-cache}}</span>`. This placeholder will be cached, and `edgestash.parse_response()` will ensure it gets recognized as an *Edgestash* placeholder.

`vcl_backend_response` also contains logic to refresh the shopping cart information when we receive the backend response for requests that add or delete shopping cart items.

The refresh happens by calling the custom `refresh_cart_memcached` subroutine. Let's have a look at this mysterious `refresh_cart` subroutine:

```
sub refresh_cart {
    if(sessions.exists("sess_" + cookieplus.get("PHPSESSID"))) {
        set beresp.http.session = sessions.read("sess_" + cookieplus.
get("PHPSESSID"));
        set beresp.http.items = regsub(beresp.http.ses-
sion,{".+s:11:"itemsInCart";i:([0-9]+);.+"},"\1");
        cart.set(cookieplus.get("PHPSESSID"),beresp.http.items);
    } else {
        cart.set(cookieplus.get("PHPSESSID"),"0");
    }
    unset beresp.http.session;
    unset beresp.http.items;
}
```

This subroutine will attach the value of the `PHPSESSID` to `sess_` and check whether the corresponding file exists on disk. If that is the case, it reads contents from the file. In our case this will be `sess_9755a8b773f76bffeda28f746ac3957e`.

And again, this is the what the session file looks like:

```
_sf2_attributes|a:2:{s:4:"cart";a:1:{i:1;i:9;}s:11:"itemsInCar-
t";i:9;}_sf2_meta|a:3:{s:1:"u";i:1611851104;s:1:"c";i:1611759335;s:1:
"l";s:1:"0";}
```

Using `regsub()` we're going to extract the value of the `itemsInCart` key. The `.+s:11:"itemsInCart";i:([0-9]+);.+` regular expression takes care of that, and the first regex capturing group contains this value. This value gets temporarily stored inside the `beresp.http.items` header, before finding its way to the `cart` *key-value store*.

Via `cart.set(cookieplus.get("PHPSESSID"),beresp.http.items)`, a key is stored per session, containing the number of items inside the shopping cart. This value will be used later by *Edgestash*. If the session file doesn't exist, we set the value to an empty string.

And finally, it's a matter of parsing the right *items-in-cart value* into the *Edgestash* placeholder:

```
sub vcl_deliver {
    if(edgestash.is_edgestash()) {
        edgestash.add_json({"{ "items-in-cart": ""}
            + cart.get(cookieplus.get("PHPSESSID"),0)
            + {"" }"});
        edgestash.execute();
    }
}
```

The end result

In the end we can store a template in cache that we can populate on-the-fly based on a placeholder. In this case, the *HTML code* of the application didn't even have the placeholder.

Thanks to *xbody*, the response body was modified, a placeholder was created, and *Edgestash* managed to parse in a value per user without having to create a cache variation per user.

We believe that this is a very powerful example of how to combine both modules, along with some other *Varnish Enterprise VMODs*.

A hard requirement for this example to work was having access to the *session files* of the application. When *Varnish* is hosted on the same machine as the origin application, that's an easy task. Otherwise shared storage would be required. But there are other, more creative ways of tackling this issue, as you will see later in this chapter.

8.3 HTTP calls

Varnish is all about *HTTP*:

- It accepts incoming *HTTP requests*.
- It returns *HTTP responses*.
- It connects to the origin and sends *backend HTTP requests*.
- It stores *backend HTTP responses* in cache.

The entire flow of *Varnish* is centered around *HTTP*, and yet this section is about making *HTTP calls* using `vmod_http`.

> `vmod_http` is a *Varnish Enterprise* module, but there is probably an open source equivalent out there in the community.

Sometimes your use case requires making *HTTP calls* to endpoints that are not related to the incoming *HTTP request* that *Varnish* is currently processing. This could be an *API call* to fetch stateful data. This could be an internal subrequest to another resource.

Let's look at some practical use cases.

8.3.1 Prefetching

A very common use case is using `vmod_http` for prefetching. This means retrieving content before it is actually requested by the client. The assumption is that the client will soon request that content, and having it in cache ahead of time will improve the user experience.

Link prefetching

Link prefetching is where the origin application requests prefetching of certain resources via a `Link` response header or a `<link>` *HTML tag*.

This mechanism is mostly used to load *CSS, JavaScript, images, favicons and web fonts* that are required by the website theme.

Here's a *VCL* example where link prefetching is done by inspecting the `Link` response header:

```
vcl 4.1;

import http;

sub vcl_recv {
    set req.http.X-prefetch = http.varnish_url("/");
}

sub vcl_backend_response {
    if(beresp.http.Link ~ "<([^>]+)>; rel=(prefetch|next)") {
        set bereq.http.X-link = regsub(beresp.http.Link,
"^.*<([^>]*)>.*$", "\1");
        set bereq.http.X-prefetch = regsub(bereq.http.X-prefetch,
"/$", bereq.http.X-link);
        http.init(0);
        http.req_copy_headers(0);
        http.req_set_url(0, bereq.http.X-prefetch);
        http.req_send_and_finish(0);
    }
}
```

Imagine that an *HTTP response* contains the following response header:

```
Link: </style.css>; rel="prefetch"
```

The *VCL example* above would extract the URL from this header, and then use it to make an asynchronous *HTTP request* via `vmod_http`.

This means that we don't wait for the response to be returned because frankly we don't really care about the response. We just want to trigger a subrequest that fetches the required resource. Eventually the client will request `/style.css`, and we assume that it will be in cache thanks to our prefetching logic.

The same can be done by parsing a `<link rel="prefetch" href="/style.css" />` *HTML tag* in the response body. These tags are designed to trigger prefetching at the browser level, but we might as well benefit from them in *Varnish* too.

Video prefetching

Varnish can also be used to accelerate video platforms, as you'll see in the next chapter.

OTT video streaming chops up encoded video files into various segments, each segment representing a couple of seconds of video playback.

A playlist file contains the endpoints of the various video segments. It may look like this:

```
/vod/video1_1.ts
/vod/video1_2.ts
/vod/video1_3.ts
/vod/video1_4.ts
/vod/video1_5.ts
```

Because of the sequential naming format, it's quite easy to guess what the URL of the next segment will be. And that's exactly what the `http.prefetch_next_url()` does.

The following *VCL example* will use this function to preload the next video segment:

```
vcl 4.1;

import http;

sub vcl_recv {
    if (req.url ~ "^/vod/video[0-9]+_[0-9+]\.ts") {
        http.init(0);
        http.req_copy_headers(0);
        http.req_set_method(0, "HEAD");
        http.req_set_url(0, http.prefetch_next_url());
        http.req_send_and_finish(0);
    }
}
```

`http.prefetch_next_url()` looks for numeric sequences and does a standard increment of one. This can be altered through the `count` argument.

The `prefix` argument defines a pattern that should be matched in the URL before considering the increment.

The `url` argument can be used to set the input URL that should be examined, whereas `url_prefix` allows you to prefix the URL with a *scheme* or *port*. If `url` or `url_prefix` are not set, `http://` is used as the scheme, `std.port(server.ip)` is used to determine the port, and `req.url` is used to determine the URL.

There's also a `base` argument, which defaults to `DECIMAL`, which sets the number system that is to be used. This is `DECIMAL`, `HEX` or `HEX_UPPER`.

The function can be run standalone, and here's an example that contains some arguments:

```
http.prefetch_next_url(prefix="test",
    url="/test1.txt",
    url_prefix="https://test.com:1234",
    count=4
);
```

Unsurprisingly, the output goes as follows:

```
https://test.com:1234/test5.txt
```

> The `http.prefetch_next_url()` can also be used outside of the video-streaming scope. Paginated web content is also a good use case.

8.3.2 API calls

It is also possible to perform *HTTP requests* to remote hosts that are not directly related to the origin platform. *API* calls could be made, and the output could be parsed in *VCL*.

Here's an example where we query a *RESTful API* to get the current weather in London:

```
vcl 4.1;

import http;

sub vcl_backend_response {
    http.init(0);
    http.req_set_header(0, "Host", "www.metaweather.com");
    http.req_set_url(0, "https://www.metaweather.com/api/loca-
tion/44418/");
    http.req_send(0);
    http.resp_wait(0);
    if (http.resp_get_status(0) != 200) {
        return (error(500,""));
    }
    json.parse(http.resp_get_body(0));
    xbody.regsub({"(<h1 class="fw-light">The weather in London: )
([^<]+)(</h1>)"},
        "\1"+json.get(".consolidated_weather[0].weather_state_
name")+"\3");

}
```

Before the object is stored in cache, a request to `https://www.metaweather.com/api/location/44418/` is made, which returns *JSON data* containing the weather in London.

`json.parse(http.resp_get_body(0))` will parse the resulting *JSON* output, which we can filter using `json.get()`. In this case we care about the `consolidated_weather` property. This is an array, and we grab the first item from that array and return the `weather_state_name` property to get the current weather.

In the end we use `xbody.regsub()` to inject the actual weather.

Imagine receiving the following *HTML tag* from the origin:

```
<h1 class="fw-light">The weather in London: Sunny</h1>
```

Our `vmod_http`, `vmod_json` and `vmod_xbody` logic will cause the following string to be stored in cache:

```
<h1 class="fw-light">The weather in London: Light Rain</h1>
```

8.3.3 Authentication

This chapter has a section dedicated to authentication, so we won't go into great detail here. We will just throw in one basic example where *Varnish* can act as an *authentication gateway*.

Imagine that you want to protect your web application with *basic authentication*. But if you remember the *built-in VCL*, a `return(pass)` will take place when an `Authorization` header is found in the request.

The idea is to terminate the authentication *on the edge* and forward the authentication request to an endpoint via `vmod_http`.

Here's an example where we offload the authentication to `https://auth.example.com/auth`:

```
vcl 4.1;

import kvstore;
import http;

sub vcl_init {
    new auth = kvstore.init();
}

sub vcl_recv {
    if (auth.get(client.ip,"0") == "0" {
        http.init(0);
        http.req_copy_headers(0);
        http.req_set_url(0, "https://auth.example.com/auth");
        http.req_send(0);
        http.resp_wait(0);
        if (http.resp_get_status(0) != 200) {
            return(synth(401,"Authentication required"));
        }
        auth.set(client.ip,"1",3h);
        unset req.http.Authorization;
    }
}

sub vcl_synth {
    if (resp.status == 401) {
        set resp.http.www-authenticate = "Basic";
    }
}
```

The `http.req_copy_headers(0)` function ensures that client request headers are forwarded to the authentication endpoint. If the `Authorization` header is missing, or does not match the expected credentials, an *HTTP 401* is triggered, which we return to the client.

If the authentication is successful, we store the authentication state inside a `vmod_kvstore` instance. We also strip off the `Authorization` header to ensure that the *built-in VCL* serves the request from cache.

As you can see, `vmod_http` offers a lot of options and will be featured again in this chapter. You'll also see a dedicated authentication section later on in this chapter.

8.4 Database access

In terms of interacting with stateful data that can be used to offer a personalized caching experience, we already used *the file system* and *API calls*.

Although they are valid candidates as the *source of truth*, there are limiting factors:

- The required files aren't always available to *Varnish*.
- Reading files may not always provide the right *data-querying facilities*.
- The source data may not be accessible via an *API*.
- The *API* containing the data may not be equipped to scale along with *Varnish*, causing a potential outage on the *API* due to excessive load.

Unless data is readily available in files, or unless *data APIs* can keep up with *Varnish*, we need to find another solution.

Having direct access to a database may be the better solution. The term *database* can refer to many implementations. Some databases may be accessible via a *RESTful API*, which can be leveraged using `vmod_http`.

In this section, we're going to cover four types of databases:

- SQLite
- Key-value storage (kvstore)
- Memcached
- Redis

8.4.1 SQLite

For the record: *SQLite* is a library that implements a serverless, self-contained relational database system. *Varnish Enterprise* contains a *VMOD* that interacts with *SQLite*. We already featured this *VMOD* in *chapters 5 and 2*.

In *chapter 5* I showed you an example where sessions were stored in the database, and that a cookie value was used to retrieve the username of a logged in user.

This time, we'll use *SQLite* to store caching policies about specific pages.

Here are the commands you need to create and populate the database:

```
sqlite3 sqlite.db <<EOF
CREATE TABLE pages (
    cache BOOLEAN NOT NULL,
    url TEXT NOT NULL,
    host TEXT NOT NULL,
    PRIMARY KEY (url, host)
);

INSERT INTO pages (cache,url,host) VALUES
(0,'/checkout','example.com'),
(1,'/','example.com'),
(1,'/products','example.com'),
(0,'/cart','example.com');
EOF
```

Once the database has been put in place, we can match the *URL* and *hostname* of a page to determine its caching behavior. When the page is not found, the *built-in VCL* behavior is used.

Here's the *VCL*:

```
vcl 4.1;

import sqlite3;

sub vcl_init {
    sqlite3.open("/etc/varnish/sqlite.db", "|;");
}

sub vcl_fini {
    sqlite3.close();
}

sub vcl_recv {
    set req.http.cache = sqlite3.exec("SELECT `cache` FROM `pages`
WHERE url='"
        + sqlite3.escape(req.url) + "' AND host='"
        + sqlite3.escape(req.http.host) + "'");
    if(req.http.cache == "1") {
        return(hash);
    } elseif (req.http.cache == "0") {
        return(pass);
    }
}
```

The output from `sqlite3.exec` is used to determine the value of the `cache` database field, based on the `url` and `hostname` values.

If there's a matching row in the database, and the `cache` field is `1`, the page is cacheable and `return(hash)` is called. If `cache` is `0`, `return(pass)` is called.

If there's no matching row, we're not returning anything, which means the *built-in VCL* behavior applies.

SQLite is a very lightweight database system and performs quite well for read-only access. As soon as you start writing to the database in *VCL*, latency will occur because write operations lock the database file.

8.4.2 Key-value storage (kvstore)

Can `vmod_kvstore` be considered a database? The examples we used throughout the book would suggest otherwise: the *key-value store* is populated in *VCL*, and a restart removes all content.

However, there is a very basic level of persistence available that can be triggered via the `.init_file()` function.

Here's the `vmod_kvstore` implementation of the *SQLite* example, but backed by a file:

```
vcl 4.1;
import kvstore;

sub vcl_init {
    new pages = kvstore.init();
    pages.init_file("/etc/varnish/pages.store",",");
}

sub vcl_recv {
    set req.http.cache = pages.get(req.http.host+req.url,"");
    if(req.http.cache == "1") {
        return(hash);
    } elseif (req.http.cache == "0") {
        return(pass);
    }
}
```

The following command can be used to populate the `pages.store` file that contains the same rules as the *SQLite* database:

```
$ cat <<EOF > /etc/varnish/pages.store
> example.com/,1
> example.com/products,1
> example.com/cart,0
> example.com/checkout,0
> EOF
```

The `pages.init_file("/etc/varnish/pages.store",",")` function can be called in other places in your *VCL* when a resynchronization is required.

This persisted *kvstore* example will perform better than *SQLite*, but does not offer the flexibility of the *SQL* language.

8.4.3 Memcached

Memcached is a *distributed key-value store* that has client implementations in many programming languages. It is extremely fast and scalable, but offers no persistence layer. Technically, *Memcached* can be viewed as simple a cache that is accessible over the network.

`vmod_memcached` is an open source *VMOD* that provides access to a *Memcached* setup. It is available via https://github.com/varnish/libvmod-memcached, but is also packaged with *Varnish Enterprise*.

Let's revisit the *basic authentication* example from earlier in this chapter. We featured this example to show the power of `vmod_http`. Let's strip out the *HTTP calls* and replace them with *Memcached calls*.

Here's the code:

```
vcl 4.1;

import crypto;
import memcached;

sub vcl_init {
   memcached.servers("--SERVER=192.168.98.101");
   memcached.error_string("error");
}

sub vcl_recv {
   if (req.http.Authorization !~ "^Basic ([a-z-A-Z0-9=]+)$") {
       return(synth(401,"Authentication required"));
   }

   set req.http.base64 = regsub(req.http.Authorization,"^Basic ([a-z-
A-Z0-9=]+)$","\1");
   set req.http.usernamepassword = crypto.string(crypto.base64_de-
code(req.http.base64));
   set req.http.username = regsub(req.http.usernamepass-
word,"^([^:]+):([^:]+)$","\1");
   set req.http.password = regsub(req.http.usernamepass-
word,"^([^:]+):([^:]+)$","\2");
   set req.http.memcached = memcached.get(req.http.username);

   if (req.http.memcached == "error") {
       return(synth(403));
   }

   if (req.http.password != req.http.memcached) {
       return(synth(401,"Authentication required"));
   }
   unset req.http.Authorization;
   unset req.http.base64;
   unset req.http.usernamepassword;
   unset req.http.username;
   unset req.http.password;
   unset req.http.memcached;
}
```

The *Memcached* server is accessible via `192.168.98.101` on the standard `11211` port and contains login credentials. *Varnish* uses these credentials to grant or deny access to the platform.

Varnish decodes the `Authorization` header using the `crypto.base64_decode()` function. Via regular expressions, the *username* and *password* are extracted.

The *Memcached* key is the username, and the corresponding value is the password. If a *Memcached* lookup results in an error, this means the user was not found. In that case we return an *HTTP 403* response.

If the passwords don't match, we return an *HTTP 401* response, which gives the client the opportunity to try logging in again.

Once authentication is successful, the `Authorization` header is stripped off to ensure the *built-in VCL* can consider the request cacheable.

Memcached can also be used to store session information, or as a way to store projected results from relational databases.

8.4.4 Redis

Redis is also a *distributed key-value store*, like *Memcached*. It can be considered the successor of *Memcached* and offers a lot more features. To some extent we can say that *Redis* is steadily becoming the industry standard.

Unlike *Memcached*, *Redis* offers multiple data types and specific commands to interact with them in an atomic way. *Redis* also offers persistence, replication, security, and many more operational features.

The fun thing about *Redis* is that it has a *LUA scripting language*, which allows you to script certain behavior.

There is an open source *VMOD* available for *Redis*, which you get via https://github.com/carlosabalde/libvmod-redis. It has a very extensive *API*.

Let's feature an example where *Redis* can be used to provide a *personalized caching experience*.

A shopping cart example

Remember the shopping cart example from earlier in this chapter? We used the file system to access the session file, and we extract the right key from the serialized session data.

It's easy to replicate this example and use *Redis* instead. However, this example will store the product and session data in a more intuitive way:

- Products will be stored as *Redis hashes* and product properties will be stored as fields for the hash.
- Shopping cart items will be stored in a *Redis list* per session.

So whenever someone adds a product to their shopping cart, an `RPUSH $sessionId $productId` command is sent to *Redis*. And whenever the quantity of a product in the cart is decreased, an `LREM $sessionId 1 $productId` is used. When a complete product is removed from the shopping cart, an `LREM $sessionId 0 $productId` command is sent to *Redis*.

Computing the number of items in the shopping cart can be done using the following *Redis* command:

```
LLEN $sessionId
```

If we have access to *Redis* from *VCL*, there are many ways we can offload this stateful logic from the origin, but in this example we'll limit it to counting the shopping cart items.

Here's the *VCL* code:

```
vcl 4.1;

import redis;
import cookieplus;
import xbody;
import edgestash;

sub vcl_init {
    new sessions = redis.db(
        location="192.168.98.102:6379",
        shared_connections=false,
        max_connections=1);
}

sub vcl_recv {
    cookieplus.keep("PHPSESSID");
    cookieplus.write();
    if(req.url ~ "^/add/to/cart/[0-9]+$" || req.url ~ "^/remove/from/
cart/[0-9]+") {
        return(pass);
    }
    if(req.url == "/") {
        return(hash);
    }
}

sub vcl_backend_response {
    if(bereq.url == "/") {
        unset beresp.http.Cache-Control;
        set beresp.ttl = 3600s;
        xbody.regsub({"(<span id="items-in-cart" [^>]+>)(\w*)(</
span>)"},
        {"\1{{items-in-cart}}\3"});
        edgestash.parse_response();
    }
}

sub vcl_deliver {
    sessions.command("LLEN");
    sessions.push(cookieplus.get("PHPSESSID"));
    sessions.execute();
    if(edgestash.is_edgestash() && sessions.reply_is_integer()) {
        edgestash.add_json({"{ "items-in-cart": ""}
            + sessions.get_integer_reply()
            + {"" }"});
        edgestash.execute();
    }
}
```

Let's talk through this one:

- In `vcl_init` we initialize a *Redis client object* called `sessions`.
- In `vcl_recv` we strip off all cookies except `PHPSESSID`.
- In `vcl_recv` we don't allow `/add/to/cart/$productId` and `/remove/from/cart/$productId` to be served from cache.
- In `vcl_recv` we explicitly cache the homepage, despite the `PHPSESSID` cookie being present.
- In `vcl_backend_response` we use `xbody.regsub()` to replace the *items in cart counter* with a `{{items-in-cart}}` *Edgestash* placeholder.
- In `vcl_deliver` we execute an `LLEN` *Redis* command to get the number of items in the shopping cart.
- In `vcl_deliver` we parse the `LLEN` *Redis* value in the `items-in-cart` placeholder.

Instead of temporarily storing the value via `vmod_kvstore`, we directly connect to *Redis* at delivery time. Although *Redis* scales really well, there might be some operational concerns. Please keep in mind that your *Redis* server should be properly tuned if you receive a lot of incoming requests.

8.5 Geo features

Another very powerful piece of information you can retrieve is the *geographical location of the client*.

Although there are *APIs* you can call using `vmod_http`, the overhead may slow us down at scale. A superior solution is to use *MaxMind's geoIP database*. This proprietary database, which has a free version, maps *IP addresses* to *geographical locations*.

There are both open source *VMODs* and proprietary *VMODs* available. They all rely on `libmaxminddb`.

As a developer, you can go to https://dev.maxmind.com/ to obtain a free version of the *geoIP database*:

- `GeoLite2-Country.mmdb`: a database that only contains country and continent information
- `GeoLite2-City.mmdb`: an extended version of the database that contains country, continent, city and geolocation information

As an administrator, it is your responsibility to keep the database up-to-date.

Being able to geographically locate the user allows you to perform *geotargeting*, but even *geoblocking*.

Geotargeting involves putting the user in a certain category based on their location. This is important when you build your own CDN because you can send users to the closest *point of presence (PoP)*. Having content as close to your users as possible will decrease latency and increase the quality of experience.

Another example of *geotargeting* is presenting localized content to the user. Many multinational corporations have separate websites per country. Being able to suggest the right one based on the *client IP address* contributes to a good user experience.

Geoblocking is used to refuse access to certain content. Websites or *OTT video platforms* that are funded with taxpayer money will refuse access to their platforms for users outside of the country.

8.5.1 vmod_geoip2

`vmod_geoip2` is an open source *VMOD* that is available on https://github.com/fgsch/libvmod-geoip2.

The `geoip2.geoip2()` function loads the *MaxMind GeoIP* database and returns an object. This object uses the `.lookup()` method to retrieve geographical information.

Here's a very simple *geoblocking* example:

```
vcl 4.1;

import geoip2;

sub vcl_init {
    new country = geoip2.geoip2("/etc/varnish/GeoLite2-Country.
mmdb");
}

sub vcl_recv {
    if(country.lookup("country/iso_code", client.ip) != "BE") {
        return(403,"Access from " + country.lookup("country/names/
en", client.ip) + " not allowed");
    }
}
```

8.5.2 vmod_mmdb

`vmod_mmdb` is a *Varnish Enterprise* module that uses the same database file provided by *MaxMind*. Here's the equivalent of the previous example:

```
vcl 4.1;

import mmdb;

sub vcl_init {
    new country = mmdb.init("/etc/varnish/GeoLite2-Country.mmdb");
}

sub vcl_recv {
    // there is a convenience function to
    // retrieve the country code directly, let's use it!
    if(country.country_code(client.ip) != "BE") {
        return(403,"Access from " + country.lookup(client.ip, "coun-
try/names/en") + " not allowed");
    }
}
```

8.5.3 Lookup filters

Both *VMODs* have a `.lookup()` method that takes a *lookup path*. This path is used to retrieve the information from the database.

The `GeoLite2-Country.mmdb` database only contains country and continent information. Here are a couple of examples of various paths:

- `continent/code`: for example `EU`
- `continent/names/en`: for example `Europe`
- `country/is_in_european_union`: for example `true`
- `country/names/iso_code`: for example `BE`
- `country/names/en`: for example `Belgium`

There is a German, Spanish, French, Japanese, Portuguese, Russian and Chinese alternative for `continent/names/en` and `country/names/en`.

Here's an overview of paths you can use to retrieve the country name in the various supported languages:

- `country/names/de`
- `country/names/en`
- `country/names/es`
- `country/names/fr`
- `country/names/ja`
- `country/names/pt-BR`
- `country/names/ru`
- `country/names/zh-CN`

These language codes are also available for continent names.

The `GeoLite2-City.mmdb` also contains all of the above but is supplemented with city and geolocation information.

Here's an overview of additional lookup filters for the city database:

- `city/names/de`
- `city/names/en`
- `city/names/es`
- `city/names/fr`
- `city/names/ja`
- `city/names/pt-BR`
- `city/names/ru`
- `city/names/zh-CN`
- `location/accuracy_radius`
- `location/latitude`
- `location/longitude`
- `location/time_zone`
- `postal/code`

There is also a `subdivision` field that describes states, provinces, or communities within a country, but we're not going to cover this in the book.

8.5.4 Backend geotargeting example

The following example looks up the *continent code* for the client IP address based on the `GeoLite2-Country.mmdb` databases, and matches this with available backends stored using `vmod_kvstore`.

If no corresponding backend is found, the default one is used.

```
vcl 4.1;

import kvstore;
import mmdb;
import std;

backend default {
    .host="default.backend.example.com";
}

backend de {
    .host="de.backend.example.com";
}

backend us {
    .host="us.backend.example.com";
}

backend br {
    .host="br.backend.example.com";
}

backend jp {
    .host="jp.backend.example.com";
}

backend sa {
    .host="sa.backend.example.com";
}

backend au {
    .host="au.backend.example.com";
}

sub vcl_init {
    new geo = mmdb.init("/etc/varnish/GeoLite2-Country.mmdb");
    new backends = kvstore.init();
    backends.set_backend("EU",de);
    backends.set_backend("NA",us);
    backends.set_backend("SA",br);
    backends.set_backend("AS",jp);
    backends.set_backend("OC",au);
    backends.set_backend("AF",sa);
}

sub vcl_recv {
    set req.backend_hint = backends.get_backend(
        geo.lookup(
            client.ip,
            "continent/code"
        ),
        default
    );
}
```

The example above features backends in Germany, the United States, Brazil, Japan, Australia and South Africa. Each of these endpoints is associated with the corresponding continent code.

If you're in Antarctica, or there is an issue mapping the IP address to a location, the default backend is used.

This same example can easily be reproduced using the open source `vmod_geoip2`.

8.6 Synthetic responses

Throughout the book, the primary focus has been caching content from the origin. For non-cacheable content we provided ways to bypass the cache.

As you have already seen, this chapter is about caching otherwise uncacheable content, and about offloading the uncacheable logic *on the edge*.

But instead of serving content from the origin, we can cut out the origin and produce the content ourselves. We do this by serving *synthetic HTTP responses*.

This is already a familiar concept by now, as `return(synth())` and `vcl_synth` have been covered a number of times.

If we have access to the stateful data, or we can compute the data ourselves, we can produce *synthetic HTTP responses* without accessing the origin server. We can do this for all content or for select endpoints.

We can produce *HTML* and basically act as a web server. We can also produce *JSON* or *XML* and become a *RESTful API application*.

In previous sections of this chapter, we talked about *file system access*, about access to *HTTP services*, and about access to *Memcached* and *Redis*. We can query these data sources and use synthetic output to visualize the data.

8.6.1 Synthetic output and no backend

If you return *synthetic output*, you don't really need to define a backend, and `backend default none;` will ensure `varnishd` doesn't complain when you load a *VCL* that doesn't have a real backend.

You can use `return(synth(200,"OK"))` to return a *synthetic response*. Any request that doesn't return synthetic output will return an `HTTP 503 Backend fetch failed` error.

The `synth()` function is very limited in its capabilities: only a status code and some text can be used, which are parsed in a pretty horrible *HTML template* :

```
<!DOCTYPE html>
<html>
  <head>
    <title>200 OK</title>
  </head>
  <body>
    <h1>Error 200 OK</h1>
    <p>OK</p>
    <h3>Guru Meditation:</h3>
    <p>XID: 32770</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
```

This is not really user-friendly unless we modify what `vcl_synth` returns.

Loading an HTML template

Using `std.fileread()` you can load an HTML file from disk, which serves as the template. Via `regsuball()`, the `<<REASON>>` placeholder can be replaced with the value of `resp.status`.

Here's an example:

```
vcl 4.1;

import std;

backend default none;

sub vcl_recv {
    return(synth(200,"Something cool"));
}

sub vcl_synth {
    if(req.url == "/") {
        set resp.http.Content-Type = "text/html";
        set resp.body = regsuball(
            std.fileread("/etc/varnish/index.html"),
            "<<REASON>>",
            resp.reason);
    } else {
        set resp.status = 404;
        set resp.http.Content-Type = "text/plain";
        set resp.body = "Not found";
    }
    return(deliver);
}
```

If you only have a couple of files to serve, this will do, and it will be very powerful. Don't forget that the output from `std.readfile()` is only processed at compile time. This means that no file system access is done at runtime.

Creating a simple API

The previous example showed some of the possibilities of synthetic responses by overriding `vcl_synth`. Let's spice it up a bit and return dynamic content.

The following example features a very small *RESTful API* that returns a *JSON object* containing the *username* and the *number of items in the shopping cart* for a session that was established.

The session is identified by a `sessionId` cookie, and the session information is stored in *Redis*.

`vmod_redis` is an open source *VMOD* by Carlos Abalde. It is not packaged with *Varnish Cache* or *Varnish Enterprise*, but you can download the source code via https://github.com/carlosabalde/libvmod-redis.

The session information is stored in a *Redis hash*, which has multiple fields. The following *Redis CLI* command returns that hash for session ID `123`:

```
127.0.0.1:6379> hgetall 123
1) "username"
2) "JohnSmith"
3) "items-in-cart"
4) "5"
```

You can see that the username for session `123` is `JohnSmith`. John has `5` items in his shopping cart.

We can create a *RESTful API* that consumes this data and returns it in `vcl_synth`:

```
vcl 4.1;

import redis;

backend default none;

sub vcl_init {
    new redis_client = redis.db(
        location="redis:6379",
        shared_connections=false,
        max_connections=1);
}

sub vcl_recv {
    return(synth(200));
}

sub vcl_synth {
    if(req.url == "/api/session") {
        set resp.http.sessionId = regsub(req.http.Cookie,"^.*;?\s*-
sessionId\s*=\s*([0-9a-zA-z]+)\s*;?.*","\1");
        redis_client.command("HMGET");
        redis_client.push(resp.http.sessionId);
        redis_client.push("username");
        redis_client.push("items-in-cart");
        redis_client.execute();
        set resp.http.Content-Type = "application/json";
        set resp.body = {"{
            "username": ""} + redis_client.get_array_reply_value(0) +
{"",
            "items-in-cart": ""} + redis_client.get_array_reply_val-
ue(1) + {""
        }"};
        return(deliver);
    }
}
```

Because the session information is stored in a *Redis hash*, a `HMGET` is required to retrieve multiple fields. The parsed command would be `HMGET 123 username items-in-cart`.

In *VCL* we can use `redis_client.get_array_reply_value()` to retrieve the value of individual fields based on an index because *Redis* returns the output as an array.

When we call the `/api/session` endpoint using the right cookie value, the output will be the following:

```
$ curl -H"Cookie: sessionId=123" localhost/api/session
{
  "username": "JohnSmith",
  "items-in-cart": "5"
}
```

8.6.2 Synthetic backends

The previous examples in this section all leveraged the `vcl_synth` subroutine to return synthetic output. Although this works fine, the output is not cached, and *ESI* or *Gzip compression* aren't supported either.

When *edge logic* depends on external services, large traffic spikes may overload these services and result in latency.

Varnish Enterprise offers *synthetic backends* through `vmod_synthbackend`: synthetic objects will be inserted at the beginning of the *fetch pipeline*, which gives them the same behavior as regular objects.

The *API* for `vmod_synthbackend` has a couple of functions:

- `synthbackend.mirror()` will mirror the request information and will return the request body into the response body.
- `synthbackend.from_blob()` will create a response body using *BLOB data*.
- `synthbackend.from_string()` will create a response body using *string data*.
- `synthbackend.none()` will return a *null backend*.

The function we're mainly interested in is `synthbackend.from_string()`. The following example is based on the previous one where *Redis* is used to return session information in a *RESTful API*.

Instead of sending a command to *Redis* for every request, the following example will cache the output and will create a cache variation per session.

Here's the code:

```
vcl 4.1;

import redis;
import synthbackend;
import ykey;

backend default {
    .host = "backend.example.com";
}

sub vcl_init {
    new redis_client = redis.db(
        location="redis:6379",
        shared_connections=false,
        max_connections=1);
}
```

```
sub vcl_recv {
    if(req.url == "/api/session") {
        set req.http.sessionId = regsub(req.http.Cookie,"^.*;?\s*ses-
sionId\s*=\s*([0-9a-zA-z]+)\s*;?.*","\1");
        if(req.method == "PURGE") {
            ykey.purge(req.http.sessionId);
        }
        return(hash);
    }
}

sub vcl_hash {
    hash_data(req.http.sessionId);
}

sub vcl_backend_fetch {
    if(bereq.url == "/api/session") {
        redis_client.command("HMGET");
        redis_client.push(bereq.http.sessionId);
        redis_client.push("username");
        redis_client.push("items-in-cart");
        redis_client.execute();
        set bereq.backend = synthbackend.from_string({"{
            "username": ""} + redis_client.get_array_reply_value(0) +
{"",
            "items-in-cart": ""} + redis_client.get_array_reply_val-
ue(1) + {""
        }"});
    } else {
        set bereq.backend = default;
    }
}

sub vcl_backend_response {
    if(bereq.url == "/api/session") {
        set beresp.http.Content-Type = "application/json";
        set beresp.ttl = 3h;
        ykey.add_key(bereq.http.sessionId);
    }
}
```

Let’s break this example down, and explain what is going on:

Requests for `/api/session` are cacheable, even *cookies* are used. The `sessionId` cookie is extracted and stored in `req.http.sessionId` for later use.

If `PURGE` requests are received for `/api/session`, `vmod_ykey` will evict objects from cache that match the *session ID*.

When we look up requests for `/api/session`, we ensure the *session ID* is used as a *cache variation*.

And when requests for `/api/session` cause a *cache miss*, a *synthetic object* is inserted into the cache via `synthbackend.from_string()`. The string contains a *JSON object* that is composed by fetching the session information from *Redis*.

Backend requests for other endpoints are sent to the *default backend*, which is not a synthetic one.

When *synthetic responses* are received for `/api/session`, the `Content-Type: application/json` response header is set and the *TTL* for the object is set to three hours.

When such a response is received, the *session ID* is registered as a key in `vmod_ykey`.

The following `curl` request will still output the personalized *JSON* response:

```
$ curl -H"Cookie: sessionId=123" localhost/api/session
{
  "username": "JohnSmith",
  "items-in-cart": "5"
}
```

The only difference is that the value is cached per session for three hours. If at any point the object needs to be updated, a `PURGE` call can be done, as illustrated below:

```
$ curl -XPURGE -H"Cookie: sessionId=123" localhost/api/session
{
  "username": "JohnSmith",
  "items-in-cart": "5"
}
```

8.7 Authentication

State often gets in the way when it comes to cacheability. In *HTTP* we usually pass information about state via a `Cookie` header or an `Authorization` header.

The *built-in VCL* is very explicit about this:

```
if (req.http.Authorization || req.http.Cookie) {
    /* Not cacheable by default */
    return (pass);
}
```

When the `Authorization` or `Cookie` request header are present, *Varnish* will not cache by default.

In earlier chapters we already explained how you can maintain cacheability without getting rid of all your cookies. In this section we'll do the same for the `Authorization` header.

This section is all about *offloading authentication* and how to turn *Varnish* into an *authentication gateway*.

8.7.1 Basic authentication

Basic authentication is pretty basic, as the name indicates: the username and password are sent as a *base64 encoded string*. Within that string, the username and password are separated by a colon.

The example below will force *basic authentication* before the page is displayed:

```
vcl 4.1;
sub vcl_recv {
    if(req.http.Authorization != "Basic YWRtaW46c2VjcmV0") {
        return (synth(401, "Restricted"));
    }
}

sub vcl_synth {
    if (resp.status == 401) {
        set resp.http.WWW-Authenticate = {"Basic realm="Restricted
area""};
    }
}
```

In this case the username is `admin` and the password is `secret`; this corresponds to the following `Authorization` header:

```
Authorization: Basic YWRtaW46c2VjcmV0
```

If the credentials don't match, an *HTTP 401* error is returned. To trigger web browsers to present a login screen when invalid credentials are received, the following response header is returned:

```
WWW-Authenticate: Basic realm="Restricted area"
```

This is a very static authentication mechanism that doesn't offer lots of flexibility, and where passwords are stored in the *VCL file*. We can do better.

Ensuring cacheability

It is important to know that even though we offload the authentication from the *origin*, the *built-in VCL* will still not allow the corresponding response to be served from cache.

We tackle this issue by stripping off the `Authorization` header when we're done offloading the authentication layer.

This is the *VCL code* you add at the end of your authentication logic:

```
unset req.http.Authorization;
```

vmod_basicauth

As mentioned in *chapter 5*, there's `vmod_basicauth`, which loads a typical `.htpasswd` file from disk. The value of the `Authorization` header can be matched against the loaded values.

The following example has already been featured but illustrates nicely how the logic is abstracted into a *VMOD*:

```
vcl 4.1;

import basicauth;

```

```
sub vcl_recv {
    if (!basicauth.match("/var/www/.htpasswd",req.http.Authoriza-
tion)) {
        return (synth(401, "Restricted"));
    }
}

sub vcl_synth {
    if (resp.status == 401) {
        set resp.http.WWW-Authenticate = {"Basic realm="Restricted
area""};
    }
}
```

Not only does this *VMOD* make offloading authentication a lot cleaner, it also supports *hashed passwords*.

The easiest way to generate the `.htpasswd` file is by using the `htpasswd` program, which is part of a typical *Apache* setup.

The example below shows how to generate a new `.htpasswd` file with credentials for the `admin` user:

```
$ htpasswd -c -s .htpasswd admin
New password:
Re-type new password:
Adding password for user admin
```

The `-c` flag will make sure the file is created, and the `-s` flag ensures *SHA hashing*.

We can then add more users to the file:

```
$ htpasswd .htpasswd thijs
New password:
Re-type new password:
Adding password for user thijs
```

This command will add the user `thijs` to the existing `.htpasswd` file using *MD5 hashing*.

Although not advised, it is also possible to add clear text passwords using the `-p` flag, as illustrated below:

```
$ htpasswd -p .htpasswd varnish
New password:
Re-type new password:
Adding password for user varnish
```

If we look inside the `.htpasswd` file, we can see the various users and corresponding password hashes:

```
admin:{SHA}W6ph5Mm5Pz8GgiULbPgzG37mj9g=
thijs:$apr1$7VbVkafq$rx9KxPEMy4bOkb61HNeY4.
varnish:cache
```

Unless the password is in clear text, the hashing algorithm is attached to the password. All these hash formats are supported by `vmod_basicauth`, which makes this a safe way to interact with passwords.

See http://man.gnu.org.ua/manpage/?3+vmod-basicauth for more information about this *VMOD*.

Hashed passwords inside vmod_kvstore

Instead of relying on `vmod_basicauth`, we can write our own logic, and we can choose our own password storage. However, we also need a way to hash passwords.

`vmod_digest` is an open source *VMOD* that can be used to create hashes in *VCL*. You can download the source from https://github.com/varnish/libvmod-digest.

This *VMOD* is also packaged in *Varnish Enterprise*. `vmod_crypto` is a competing *VMOD* that is only available in *Varnish Enterprise*. However, in this section, I'll only focus on `vmod_digest`.

By storing these hashed passwords inside `vmod_kvstore`, we have quick access to the credentials. Because `vmod_kvstore` stores data in memory, one would think that the hashed passwords cannot be persisted on disk. Luckily the `.init_file()` method allows us to preload the *key-value store* with that coming from a file.

Here's the *VCL code*:

```
vcl 4.1;

import kvstore;
import digest;

sub vcl_init {
    new auth = kvstore.init();
    auth.init_file("/etc/varnish/auth",":");
}

sub vcl_recv {
    if(req.http.Authorization !~ "^Basic .+$") {
        return(synth(401,"Authentication required"));
    }
    set req.http.userpassword = digest.base64url_decode(
        regsub(req.http.Authorization,"^Basic (.+)$","\1")
    );
    set req.http.user = regsub(req.http.userpass-
word,"^([^:]+):([^:]+)$","\1");
    set req.http.password = regsub(req.http.userpass-
word,"^([^:]+):([^:]+)$","\2");

    if(digest.hash_sha256(req.http.password) != auth.get(req.http.
user,"0")) {
        return(synth(401,"Authentication required"));
    }

    unset req.http.user;
    unset req.http.password;
}

sub vcl_synth {
    if (resp.status == 401) {
        set resp.http.WWW-Authenticate = {"Basic realm="Restricted
area""};
    }
}
```

When the configuration is loaded, `vmod_kvstore` will load its contents from `/etc/varnish/auth`. Here is what this file could look like:

```
admin:5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
thijs:4e5d73505c74a4d6c80d7fe4c7b53ddb9563488ee9f2e91200a78413f86e2597
```

The usernames, which are the keys, appear first on each line. The passwords are hashed using the *SHA256 hashing algorithm* and are delimited from the key via a colon.

The `regsub()` function helps us extract the username and password from the `Authorization` header, and the `digest.hash_sha256()` function will create the right hash.

The username is temporarily stored in `req.http.user`, and the password in `req.http.password`. In the end the value of `auth.get(req.http.user,"0")` is compared to the hashed password. If the values match, access is granted.

In this example we use `vmod_kvstore`, but `vmod_redis`, or `vmod_memcached`, or even `vmod_sqlite3` could also be viable candidates.

8.7.2 Digest authentication

Although *basic authentication* is one of the most common forms of authentication, there are some concerns: even if the stored passwords are hashed, the user does send the username and password over the wire unencrypted.

Base64 encoding is not human-readable, but it is very easy to decode. This concern is usually mitigated by *TLS* because the request is encrypted.

Digest authentication does not send clear-text passwords, but instead hashes the response. There are even more security mechanisms in place to avoid replay attacks.

Digest authentication exchange

It all starts when a server requests authentication by sending the following response:

```
HTTP/1.1 401 Authentication required
WWW-Authenticate: Digest realm="varnish",
        qop="auth",
        nonce="5f9e162f49a7811049b2d4bdf2d30196",
        opaque="c23bd2a0047189e89aa9bea67adbc1f0"
```

The *HTTP 401* indicates that authentication is required before access is allowed to the resource. The *HTTP response* provides more context by issuing a `WWW-Authenticate` header containing the digest information.

It's not as simple as requesting that the client sends a username and a password. The following information is presented by the `WWW-Authenticate` header:

- `Digest` indicates that digest authentication is required.
- `realm="varnish"` indicates that `varnish` is the realm for which valid access credentials should be provided.
- `qop="auth"` is the quality of protection and is set to `auth`.
- `nonce` is a unique value that changes for every request. It is used to avoid replay attacks.
- `opaque` is a unique value that is static.

These fields are used by the client to compose the right `Authorization` header.

Here's an example of the corresponding `Authorization` header:

```
Authorization: Digest username="admin",
    realm="varnish",
    nonce="f378c7d8a10a8ade3213fd5877b0c47d", uri="/",
    response="5bb85448beebdc6ec83c2e5712b5fdd0",
    opaque="c23bd2a0047189e89aa9bea67adbc1f0",
    qop=auth,
    nc=00000002,
    cnonce="fdd97488004e64a7"
```

As you can see, the password is not sent in clear text, but instead is part of the `response` hash.

Let's break down the entire header:

- We start with `Digest` to confirm that the authentication type is indeed *digest authentication*.
- The first field is the `username` field, which is sent in clear text. The same applies to the `realm` field.
- The `nonce` and the `opaque` fields are sent back to the server unchanged.
- The `qop` field is still set to `auth`, which confirms the *quality of protection*.
- The `response` field contains a hashed version of the password, along with some other data.
- The `nc` field is a counter that is incremented for every authentication attempt.
- The `cnonce` field is a *client nonce*

The password that is stored on the server is an *MD5 hash* of the *username*, the *realm*, and the *password*. This is how it is composed:

```
md5(username:realm:password)
```

The response that is received from the client should be matched to a server-generated response that is composed as follows:

```
hash1 = md5(username:realm:password)
hash2 = md5(request method:uri)
response = md5(hash1:nonce:nc:cnonce:qop:hash2)
```

If the `response` field sent by the client as a part of the `Authorization` header matches the response generated on the server, the user is allowed to access the content.

Offloading digest authentication in Varnish

The following example features *digest authentication offloading* in *Varnish*. The hashed passwords are stored in *Redis*.

Admittedly, it's quite a lengthy example, but there are a lot things to check when performing *digest authentication*!

```
vcl 4.1;

import redis;
import digest;
import std;

sub vcl_init {
    new redis_client = redis.db(
        location="redis:6379",
        shared_connections=false,
        max_connections=1);
}

sub vcl_recv {
    set req.http.auth-user = regsub(req.http.Authorization,{"^Digest
username="(\w+)",.*$"},"\1");
    set req.http.auth-realm = regsub(req.http.Authorization,{".*,
realm="(varnish)",.*$"},"\1");
    set req.http.auth-opaque = regsub(req.http.Authorization,{".*,
opaque="(c23bd2a0047189e89aa9bea67adbc1f0)",.*$"},"\1");
    set req.http.auth-nonce = regsub(req.http.Authorization,{".*,
nonce="(\w+)",.*$"},"\1");
    set req.http.auth-nc = regsub(req.http.Authorization,{".*,
nc=([0-9]+),.*$"},"\1");
```

```
    set req.http.auth-qop = regsub(req.http.Authorization,{".*,
qop=(auth),.*$"},"\1");
    set req.http.auth-response = regsub(req.http.Authorization,{".*,
response="(\w+)",.*$"},"\1");
    set req.http.auth-cnonce = regsub(req.http.Authorization,{".*,
cnonce="(\w+)"$"},"\1");

    if(req.http.Authorization !~ "^Digest .+$" ||
        req.http.auth-realm  != "varnish" ||
        req.http.auth-opaque  != "c23bd2a0047189e89aa9bea67adbc1f0")
{
        return(synth(401,"Authentication required"));
    }

    redis_client.command("GET");
    redis_client.push("user:" + req.http.auth-user);
    redis_client.execute();
    if(redis_client.reply_is_nil()){
        return(synth(401,"Authentication required"));
    }
    set req.http.auth-password = redis_client.get_string_reply();

    set req.http.response = digest.hash_md5(
        req.http.auth-password + ":" +
        req.http.auth-nonce + ":" +
        req.http.auth-nc + ":" +
        req.http.auth-cnonce + ":" +
        req.http.auth-qop + ":" +
        digest.hash_md5(req.method + ":" + req.url)
    );

    if(req.http.auth-response != req.http.response) {
        return(synth(401,"Authentication required"));
    }
}

sub vcl_synth {
    if (resp.status == 401) {
        set resp.http.WWW-Authenticate = {"Digest realm="varnish",
        qop="auth",
        nonce=""} + digest.hash_md5(std.random(1, 90000000)) + {"",
        opaque="c23bd2a0047189e89aa9bea67adbc1f0""};
    }
}
```

Whenever access is not granted, we return `return(synth(401,"Authentication required"));`, which triggers `vcl_synth`. Inside `vcl_synth`, we return the `WWW-Authenticate` header containing the necessary fields.

The `nonce` is different for every request. A `uuid` would be suitable for this, but only *Varnish Enterprise* has a `uuid` generator. Since this is an example that also works in *Varnish Cache*, we generated a random number and hashed it via *MD5*. `digest.hash_md5(std.random(1, 90000000))` is what we use to get that done.

In `vcl_recv` we use `regsub()` to extract the value of every field in the `Authorization` header. In the first if-statement we check whether the `Authorization` header starts with `Digest`. If not, we request reauthentication by returning the *HTTP 401* status that includes the `WWW-Authenticate` header.

The same *HTTP 401* is returned when the `realm` or `opaque` field doesn't match the expected values.

The next step involves checking if the supplied username exists in the database. In the case of the `admin` user, we perform a `GET user:admin` command in *Redis*. If *Redis* responds with a `nil` value, we can conclude that the user doesn't exist.

If *Redis* returns a string value, the value corresponds to the hashed password. This value is stored in *VCL* for later use.

Despite all these earlier checks, we still need to match the `response` field to the response that was generated. As explained earlier, we need to create a series of *MD5 hashes*:

- The password hash that comes from *Redis*. This hash is generated using the username, the realm and the password.
- A hash that contains the *request method* and *request URL*
- A response hash that uses the previous two hashes and some of the fields that were supplied by the client

In the *authentication exchange* subsection, we illustrated this using the following formula:

```
hash1 = md5(username:realm:password)
hash2 = md5(request method:uri)
response = md5(hash1:nonce:nc:cnonce:qop:hash2)
```

In the *VCL example*, the following code is responsible for creating the response hash:

```
set req.http.response = digest.hash_md5(
    req.http.auth-password + ":" +
    req.http.auth-nonce + ":" +
    req.http.auth-nc + ":" +
    req.http.auth-cnonce + ":" +
    req.http.auth-qop + ":" +
    digest.hash_md5(req.method + ":" + req.url)
);
```

And eventually the value of `req.http.response` is matched with the `response` field from the `Authorization` header. If these values match, we know the user supplied the correct credentials.

8.7.3 JSON web tokens

We often associate authentication with usernames and passwords. While these sorts of credentials are prevalent, there are also other means of authentication. Token-based authentication is one of them.

JSON web tokens (JWT) is an implementation of token-based authentication where the token contains a collection of public claims, and where security is guaranteed through a cryptographic signature.

Here's an example *JWT*:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0aGlqcyIsImV4c-
CI6MTYxNDI2NDI3MSwiaWF0IjoxNjE0MjU3MDcxLCJuYmYiOjE2MTQyNTcwNzF9.vu-
JEQOqS3uTeKFihFehiqzLVOjT7F0J8ZpIeOvEOgZc
```

It might look like gibberish, but it does make perfect sense: a *JWT* is a group of *base64 URL encoded JSON strings* that are separated by dots.

This is the composition of a *JWT*:

- The first group represents the header and contains contextual information.
- The second group is the payload: it contains a collection of public claims.
- The third group is the signature that guarantees the security and integrity of the token.

JSON web tokens are mostly used for *API authentication* and are transported as a *bearer authentication token* via the `Authorization` request header:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzd-
WIiOiJ0aGlqcyIsImV4cCI6MTYxNDI2NDI3MSwiaWF0IjoxNjE0MjU3MDcxLCJuYmYiO-
jE2MTQyNTcwNzF9.vuJEQOqS3uTeKFihFehiqzLVOjT7F0J8ZpIeOvEOgZc
```

Not only does this token serve as an authentication mechanism, it also serves as client-side session storage because the relevant client data is part of the token.

JWT header

Here's the decoded version of the header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The `alg` property refers to the algorithm that is used to sign the *JWT*. In this case this is a *SHA256-encoded HMAC signature*. The `typ` property refers to the token type. In this case it's a *JWT*.

`HS256` involves symmetric encryption. This means that both the issuer of the token and the validator of token use the same private key.

Asymmetric encryption is also supported: by using `RS256` as the value of the `alg` field. When using `RS256`, the *JWT* will be signed using a *private key*, and the *JWT* can later be verified using the *public key*.

When the application that is processing the *JWT* is the same as the one issuing the *JWT*, `HS256` is a good option. When the *JWT* is issued by a third-party application, `RS256` makes more sense. The key information would in that case meet the *JSON Web Key (JWK)* specification, which is beyond the scope of this book.

JWT payload

This is the *JSON* representation of the decoded payload:

```
{
  "sub": "thijs",
  "exp": 1614264271,
  "iat": 1614257071,
  "nbf": 1614257071
}
```

The payload's properties deliberately have short names: the bigger the property names and values, the bigger the size of the *JWT*, and the bigger the data transfer. This payload example features some *reserved claims*:

- `sub`: the *subject* of the *JWT*. This claim contains the username.
- `exp`: the *expiration* time of the token. The `1614264271` value is a Unix timestamp.
- `iat`: the *issued at time* of the token. This token was issued at `1614257071`, which is also a Unix timestamp.
- `nbf`: the *not before time* of the token is a Unix timestamp that dictates when the *JWT* becomes valid. This is also `1614257071`.

If you subtract the `iat` value from the `exp` value, you get `7200`. This represents the *TTL* of the token, which is *two hours*. The `iat` and `nbf` values are identical. This means the token is valid immediately after issuing.

You can also add your own claims to the payload. Just remember: the more content, the bigger the token, the bigger the transfer.

> Remember that the payload is not encrypted: it's just *base64 URL encoded JSON* that can easily be decoded by the client. This means that a *JWT* should not contain sensitive data that the user is not privy to.

JWT signature

The third part of the *JWT* is the signature. This signature is based on the header and payload, which means it ensures that the data is not tampered with.

When the `HS256` algorithm is used, an *HMAC signature* is generated using the *SHA256-hashing algorithm*. This signature is based on a secret key.

In the example below, the signature is generated for `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0aGlqcyIsImV4cCI6MTYxNDI2NDI3MSwiaWF0IjoxNjE0MjU3MDcxLCJuYmYiOjE2MTQyNTcwNzF9` with `supersecret` as the secret key:

```
#!/usr/bin/env bash
JWT_HEADER="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9"
JWT_PAYLOAD="eyJzdWIiOiJ0aGlqcyIsImV4cCI6MTYxNDI2NDI3MSwiaWF0I-
joxNjE0MjU3MDcxLCJuYmYiOjE2MTQyNTcwNzF9"
SECRET="supersecret"
echo -n "${JWT_HEADER}.${JWT_PAYLOAD}" \
| openssl dgst -sha256 -hmac $SECRET -binary \
| base64  | sed s/\+/-/g | sed ‘s/\//_/g’ | sed -E s/=+$//
```

When you run this script, the output would be `vuJEQOqS3uTeKFihFehiqzLVOjT-7F0J8ZpIeOvEOgZc`, which matches the *JWT* mentioned above. Don't forget that this signature is *base64 URL encoded*.

Issuing processing `HS256` tokens is done using the same *HMAC* signature, but when we use `RS256` tokens, things are slightly different.

For `RS256` tokens, the token is issued with a signature that was signed using the private key. When processing this token, the public key must be used for verification.

Here's how to create and verify a `RS256` *JWT*.

The first step is to generate the key pair:

```
#!/usr/bin/env bash
ssh-keygen -q -t rsa -b 4096 -m PEM -f private.key -P "" <<<y 2>&1 >/
dev/null
openssl rsa -in private.key -pubout -outform PEM -out public.key
```

The following script will use the `private.key` file to create the signature.

Although the *JWT header* looks the same as in the previous example, it differs. Because the `alg` property was changed from `HS256` to `RS256` for this example, we have a different header.

```
#!/usr/bin/env bash
JWT_HEADER="eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9"
JWT_PAYLOAD="eyJzdWIiOiJ0aGlqcyIsImV4cCI6MTYxNDI2NDI3MSwiaWF0I-
joxNjE0MjU3MDcxLCJuYmYiOjE2MTQyNTcwNzF9"
echo -n "${JWT_HEADER}.${JWT_PAYLOAD}" \
| openssl dgst -sha256 -sign private.key -binary \
| base64  | sed s/\+/-/g | sed ‘s/\//_/g’ | sed -E s/=+$//
```

The signature is a lot bigger for `RS256` as you can see:

```
oAVLTpK3BRny_kA40h8asQlSNENvo-xrx6_6EooM6co812AUC_agTaVQb9KIjnlVl-
9jMXdGZGFfL6pnMI4tXmZvukMonZKoEcrT8XilNRq0LUutnymObmYWY3eiQTwuQ6D-
1QPy_ykLtw78e8zig1ihLAcXp2QGwTOsc5ndMYiovCs-_zWDJoAyzy6RtbnGo7BAO8fu_
XTYKLHZAeB2ZPiVCr3mMn6H3PTJvW3PhwPyrpHQRAPX21zXP-hYDcrly-UnKIpR9qSt-
PIhPAUznrdDzZJIGvBeN_6BaShXXsze2XOE8JO-M8RUMUQ4OS8ufNo8wDxYH-C9h-
VslVlAmVqcNpc23Dtu3-k4K30ZLmINrBVFcdOHEliz93msZVIcdNDJVLZZia-JsQL-
CeNEkouiH1wLHkZYmaJLuv-dvIqOBzjMPDGbti2p1vfAjPJHAIZIRyZnfM461L01WbW-
Z7xr4hIHmQ0X5xR7_jv5rsjl2kfRlQa_JqKr9PgXPqiQ1UTvzT0O0249hjbZ7N5oo6UE-
Pd-Bi1wO9PeEjJXg75ZLBsXdoSBmvgYkceMgxvK0Lq1STw3I9HTbk26ygvqqKDo-CG-
Cv1N95ebsl3v9TTaSb4y6QLqgH7Sr3VvrdAa7NtcsSL5bVR23oJSW1P7atWBJNuC0HAx-
G5h-GffkNFSJOvml-ss
```

The end result is the following *JWT*:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0aGlqcyIsImV4c-
CI6MTYxNDI2NDI3MSwiaWF0IjoxNjE0MjU3MDcxLCJuYmYiOjE2MTQyNTcwNzF9.
oAVLTpK3BRny_kA40h8asQlSNENvo-xrx6_6EooM6co812AUC_agTaVQb9KIjnlVl-
9jMXdGZGFfL6pnMI4tXmZvukMonZKoEcrT8XilNRq0LUutnymObmYWY3eiQTwuQ6D-
1QPy_ykLtw78e8zig1ihLAcXp2QGwTOsc5ndMYiovCs-_zWDJoAyzy6RtbnGo7BAO8fu_
XTYKLHZAeB2ZPiVCr3mMn6H3PTJvW3PhwPyrpHQRAPX21zXP-hYDcrly-UnKIpR9qSt-
PIhPAUznrdDzZJIGvBeN_6BaShXXsze2XOE8JO-M8RUMUQ4OS8ufNo8wDxYH-C9h-
VslVlAmVqcNpc23Dtu3-k4K30ZLmINrBVFcdOHEliz93msZVIcdNDJVLZZia-JsQL-
CeNEkouiH1wLHkZYmaJLuv-dvIqOBzjMPDGbti2p1vfAjPJHAIZIRyZnfM461L01WbW-
Z7xr4hIHmQ0X5xR7_jv5rsjl2kfRlQa_JqKr9PgXPqiQ1UTvzT0O0249hjbZ7N5oo6UE-
Pd-Bi1wO9PeEjJXg75ZLBsXdoSBmvgYkceMgxvK0Lq1STw3I9HTbk26ygvqqKDo-CG-
Cv1N95ebsl3v9TTaSb4y6QLqgH7Sr3VvrdAa7NtcsSL5bVR23oJSW1P7atWBJNuC0HAx-
G5h-GffkNFSJOvml-ss
```

Verifying the RS256 signature requires using the public.key file, as illustrated in the script below:

```
#!/usr/bin/env bash
JWT_HEADER="eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9"
JWT_PAYLOAD="eyJzdWIiOiJ0aGlqcyIsImV4cCI6MTYxNDI2NDI3MSwiaWF0I-
joxNjE0MjU3MDcxLCJuYmYiOjE2MTQyNTcwNzF9"
JWT_SIGNATURE="oAVLTpK3BRny_kA40h8asQlSNENvo-xrx6_6EooM6co812AUC_ag-
TaVQb9KIjnlVl9jMXdGZGFfL6pnMI4tXmZvukMonZKoEcrT8XilNRq0LUutnymObmY-
WY3eiQTwuQ6D1QPy_ykLtw78e8zig1ihLAcXp2QGwTOsc5ndMYiovCs-_zWDJoAyz-
y6RtbnGo7BAO8fu_XTYKLHZAeB2ZPiVCr3mMn6H3PTJvW3PhwPyrpHQRAPX21zXP-hY-
Dcrly-UnKIpR9qStPIhPAUznrdDzZJIGvBeN_6BaShXXsze2XOE8JO-M8RUMUQ4O-
S8ufNo8wDxYH-C9hVslVlAmVqcNpc23Dtu3-k4K30ZLmINrBVFcdOHEliz93msZVIcd-
NDJVLZZia-JsQLCeNEkouiH1wLHkZYmaJLuv-dvIqOBzjMPDGbti2p1vfAjPJHAI-
ZIRyZnfM461L01WbWZ7xr4hIHmQ0X5xR7_jv5rsjl2kfRlQa_JqKr9PgXPqiQ1UTvz-
T0O0249hjbZ7N5oo6UEPd-Bi1wO9PeEjJXg75ZLBsXdoSBmvgYkceMgxvK0Lq1STw3I-
9HTbk26ygvqqKDo-CGCv1N95ebsl3v9TTaSb4y6QLqgH7Sr3VvrdAa7NtcsSL5bVR23o-
JSW1P7atWBJNuC0HAxG5h-GffkNFSJOvml-ss"
MOD=$(($(echo -n "$JWT_SIGNATURE" | wc -c) % 4))
```

```
PADDING=$(if [ $MOD -eq 2 ]; then echo -n '=='; elif [ $MOD -eq 3 ];
then echo -n '=' ; fi)
echo -n "${JWT_SIGNATURE}${PADDING}" | sed s/\-/+/g | sed 's/_/\//g'
| base64 -d > signature.rsa
echo -n "${JWT_HEADER}.${JWT_PAYLOAD}" | openssl dgst -sha256 -verify
public.key -signature signature.rsa
```

If the script ran successfully, the output will be `Verified OK`. If not, you'll get `Verification Failure`.

This script will store the *base64 URL decoded* signature in the `signature.rsa` file, and will be loaded along with the `private.key` file to perform the verification.

Long story short: the signature ensures the integrity of the payload and prevents users from getting unauthorized access because of manipulated payload.

vmod_jwt

Enough about issuing and verifying *JWT* in Bash, time to bring *Varnish* back into the picture.

Varnish Enterprise has a *VMOD* for reading and writing *JWTs*. It's called `vmod_jwt`, and here's an example of how it is used to verify the validity of a *bearer authentication token*:

```
vcl 4.1;

import jwt;

sub vcl_init {
    new jwt_reader = jwt.reader();

sub vcl_recv {
    if (!jwt_reader.parse(regsub(req.http.Authorization,"^Bearer
(.+)$","\1")) ||
        !jwt_reader.set_key("supersecret") ||
        !jwt_reader.verify("HS256")) {
        return (synth(401, "Invalid token"));
    }
}
```

First we check if the `Authorization` header contains the `Bearer` type and the payload. The next step involves setting the secret key, which is `supersecret` in this case. And finally we verify the content of the token.

The verification involves multiple steps:

- Does the *JWT* header have an `alg` property that is set to `HS256`?
- Does the *HMAC signature* using the secret key match the one we received in the *JWT*?
- Does the value of the `nbf` claim allow us to already use the token?
- If we compare the `iat` and `exp` claims, can we conclude that the token has expired?

If any of these criteria doesn't apply, the *VCL example* will return an *HTTP 401* error.

This example assumes that the *JWT* was issued by the origin, which is a common use case. The next example will completely offload authentication and will also issue *JSON web tokens*:

```
vcl 4.1;

import jwt;
import json;
import xbody;
import kvstore;
import std;
import crypto;

sub vcl_init {
    new jwt_reader = jwt.reader();
    new jwt_writer = jwt.writer();
    new auth = kvstore.init();
    auth.init_file("/etc/varnish/auth",":");
    new keys = kvstore.init();
}

sub vcl_recv {
    if(req.url == "/auth" && req.method == "POST") {
        std.cache_req_body(1KB);
        set req.http.username = regsub(xbody.get_req_body(),"^user-
name=([^&]+)&password=(.+)$","\1");
        set req.http.password = regsub(xbody.get_req_body(),"^user-
name=([^&]+)&password=(.+)$","\2");
        if(auth.get(req.http.username) != crypto.hex_encode(crypto.
hash(sha256,req.http.password))) {
            return (synth(401, "Invalid username & password"));
        }
        return(synth(700));
    }
    if (!jwt_reader.parse(regsub(req.http.Authorization,"^Bearer
(.+)$","\1"))) {
        return (synth(401, "Invalid token"));
    }

    if(!jwt_reader.set_key(keys.get(jwt_reader.to_string())) || !jwt_
reader.verify("HS256")) {
        return (synth(401, "Invalid token"));
    }
}

sub create_jwt {
    jwt_writer.set_alg("HS256");
    jwt_writer.set_typ("JWT");
    jwt_writer.set_sub(req.http.username);
    jwt_writer.set_iat(now);
    jwt_writer.set_duration(2h);
    set resp.http.key = crypto.uuid_v4();
    set resp.http.jwt = jwt_writer.generate(resp.http.key);
    keys.set(resp.http.jwt,resp.http.key);
    unset resp.http.key;
}

sub vcl_synth {
    set resp.http.Content-Type = "application/json";
    if(resp.status == 700) {
        set resp.status = 200;
        set resp.reason = "OK";
        call create_jwt;
        set resp.body = "{" + {""token": ""} + resp.http.jwt + {"""}
+ "}";
    } else {
        set resp.body = json.stringify(resp.reason);
    }
    unset resp.http.jwt;
    return(deliver);
}
```

Let’s break this one down because there’s a lot more information in this example.

The /auth endpoint that this example provides is used to authenticate users with a username and a password. These credentials are loaded into a *key-value store* but are backed by the /etc/varnish/auth file, as highlighted below:

```
new auth = kvstore.init();
auth.init_file("/etc/varnish/auth",":");
```

This is what the file looks like:

```
admin:5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
thijs:4e5d73505c74a4d6c80d7fe4c7b53ddb9563488ee9f2e91200a78413f86e2597
```

The passwords are *SHA256* hashes.

Here's the *VCL* code that performs the authentication:

```
if(req.url == "/auth" && req.method == "POST") {
    std.cache_req_body(1KB);
    set req.http.username = regsub(xbody.get_req_body(),"^user-
name=([^&]+)&password=(.+)$","\1");
    set req.http.password = regsub(xbody.get_req_body(),"^user-
name=([^&]+)&password=(.+)$","\2");
    if(auth.get(req.http.username) != crypto.hex_encode(crypto.
hash(sha256,req.http.password))) {
        return (synth(401, "Invalid username & password"));
    }
    return(synth(700));
}
```

It acts upon *HTTP POST* calls to the `/auth` endpoint and extracts the `username` and `password` fields from the *POST data*. Via `auth.get()` the username is matched to the content of the *key-value store*. The password that was received is hashed using the *SHA256-hashing algorithm*.

If the credentials don't match, an *HTTP 401* error is returned; if there is a match, some custom logic is executed inside `vcl_synth`. Because of the custom `700` status code, `vcl_synth` knows it needs to issue a token.

Here is the content of `vcl_synth`:

```
sub vcl_synth {
    set resp.http.Content-Type = "application/json";
    if(resp.status == 700) {
        set resp.status = 200;
        set resp.reason = "OK";
        call create_jwt;
        set resp.body = "{" + {""token": ""} + resp.http.jwt + {"""}
+ "}";
    } else {
        set resp.body = json.stringify(resp.reason);
    }
    unset resp.http.jwt;
    return(deliver);
}
```

The output for synthetic responses has the `application/json` content type and is formatted as a *JSON string*. When the incoming status code is `700`, we intercept the request, change the status to `200`, and return a *JSON object* that contains the *JWT*.

The custom `create_jwt` subroutine is in charge of the token creation and sends the token to the `resp.http.jwt` header that is used in `vcl_synth`.

Here's the content of `create_jwt`:

```
sub create_jwt {
    jwt_writer.set_alg("HS256");
    jwt_writer.set_typ("JWT");
    jwt_writer.set_sub(req.http.username);
    jwt_writer.set_iat(now);
    jwt_writer.set_duration(2h);
    set resp.http.key = crypto.uuid_v4();
    set resp.http.jwt = jwt_writer.generate(resp.http.key);
    keys.set(resp.http.jwt,resp.http.key);
    unset resp.http.key;
}
```

As you can see, this subroutine creates a *JWT* using the *JWT writer object* that was instantiated in `vcl_init`.

Here's what happens:

- The `alg` property of the header is set to `HS256`.
- The `typ` property of the header is set to `JWT`.
- The `sub` claim is set to the username of the logged-in user.
- The `iat` claim is set to the current timestamp.
- The `exp` claim is set to a timestamp two hours in the future.
- A *UUID* is generated and used as the secret key for the *HMAC signature*.
- This *UUID* is stored in the `keys` *key-value store*, which is used later for verification purposes.

The following `curl` command can be used to generate the token:

```
$ curl -XPOST -d"username=thijs&password=feryn" https://localhost/
auth
{"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0aGlqcy-
IsImV4cCI6MTYxNDM0MjIxMCwiaWF0IjoxNjE0MzM1MDEwLCJuYmYiOjE2MTQzMzUw-
MTB9.S5tqkGjUIJD9sTW8n0Zf9UAXIbPK_3-wCCGVP8wQSg4"}
```

A final piece of *VCL* we want to cover on *JWT* is the key verification. Here's the line of code that sets the secret key and verifies it:

```
if(!jwt_reader.set_key(keys.get(jwt_reader.to_string())) || !jwt_
reader.verify("HS256")) {
    return (synth(401, "Invalid token"));
}
```

Remember the *UUID* that was used as the secret key to sign the *JWT* in the `create_jwt` subroutine? That *UUID* was stored in a *key-value store* `keys.set()`. This means that every token has a unique secret key.

At the validation level in `vcl_recv`, we now need to fetch that secret key again via `keys.get()`. The way that secret key is identified in the *key-value store* is through the *JWT*. By putting `jwt_reader.set_key(keys.get(jwt_reader.to_string()))` in the code, we fetch the entire *JWT* string, we use it as the key in the *key-value store*, and whatever comes out is the secret key of the *HMAC signature*.

8.7.4 OAuth

OAuth is an authentication standard that delegates the processing of login credentials to a trusted third party. A typical example is the *login with Google* button that you see on many websites.

Delegating authentication to a third party results in not having to create a user account with separate credentials on each website. It's also a matter of trust: the application that wants you to log in will never have your password. This is part of the delegation process.

The concept uses a series of redirections and callbacks to exchange information:

- The first step involves redirecting the user to the login page, along with some metadata about the requesting application and requested data.
- When the login is successful, and depending on the *OAuth* request, the service will return a code.
- This code is attached to a *callback URL*, which brings the request back to the main application.
- Using that code, the application will request a set of tokens from the *OAuth service*.
- The tokens that are returned by the *OAuth service* may contain the request user information or allow access to other *APIs* that are provided by this service.

In the case of *Google's OAuth service*, you receive an *access token* and an *ID token*:

- The *access token* can grant you access to other *Google APIs*.
- The *ID token* is a *JWT* that contains the request profile information in a collection of claims.

Google OAuth in Varnish

If you look at what you need to offload *OAuth* in *Varnish*, it's not that complicated:

- You need an HTTP client. `vmod_http` can take care of that.
- You need to store some settings. We use `vmod_kvstore` to store those values.
- You need to parse *JSON* and handle *JWT*s. `vmod_json` and `vmod_jwt` are the obvious candidates.

And of course there's a *VCL* example that showcases *Varnish Enterprise*'s *OAuth* capabilities using a collection of *VMOD*s. However, this example has more than 200 lines of code. This is not practical.

My colleague Andrew created the necessary logic, which is available via https://gist.github.com/andrewwiik/3dcb9c028b15bf359ae1053b8e8f94b9.

In your *VCL* configuration, it's just a matter of including that file, overriding the necessary parameters, and calling `gauth_check` in `vcl_recv`. The rest happens automatically.

Here's the code that overrides the settings, includes the `gauth.vcl` file, and runs the *Google OAuth* logic:

```
vcl 4.1;

include "gauth.vcl";

sub vcl_init {
    gauth_config.set("client_id", "my-client-id");
    gauth_config.set("client_secret", "my-client-secret");
    gauth_config.set("callback_path", "/api/auth/google/callback");
    gauth_config.set("auth_cookie", "auth_cookie");
    gauth_config.set("signing_secret", "supersecret");
    gauth_config.set("scope", "email");
    gauth_config.set("allowed_domain", "my-domain.com");
}


sub vcl_recv {
  call gauth_check;
}
```

Let's quickly go over the various configuration parameters:

- `client_id` is the client ID for the *OAuth client* you configured for your project in the *Google API console*.
- `client_secret` is the corresponding client secret for the client ID.
- `callback_path` is the callback that is triggered when *Google's OAuth service* responds back with a code.
- `auth_cookie` is the cookie that *Varnish* will use to store the *JWT*.
- `signing_secret` is the secret key that *Varnish* will use to sign the *JWT*.
- `scope` is the scope of the *OAuth request*. In this case only the email address is requested.
- `allowed_domain` refers to the domain that the email address should have.

Don't forget to configure the allowed callback URLs in the *Google API console*. Otherwise the redirect to the callback URL will not be allowed. The hostname for this callback URL is the hostname that was used for the initial HTTP request. Also keep in mind that these are `https://` URLs.

8.8 Summary

In real-world scenarios in which cookies are omnipresent, and where authentication is sometimes required, *Varnish* can take on a much bigger role than you might think.

VCL is not only there to decide what and what not to cache; this chapter has proven that *VCL* is at the heart of *decision-making on the edge*.

Hole punching techniques such as *ESI* and *AJAX* are common practices these days. They chop up a single HTTP response into a main response and a number of fragments, each with their own *VCL* behavior and *TTL*.

Although this is a great improvement in comparison to the `return(pass)` behavior that would otherwise occur when cookies are present, the non-cacheable subrequests still have to access the origin.

This chapter has proven that *Varnish* can perform some basic logic and interact with third-party party systems.

VMODs like `vmod_xbody`, and `vmod_edgestash` have proven to be excellent utilities for changing the response body. `vmod_http` is like a Swiss army knife that offers many options to interact with external systems.

The *VCL examples* in this chapter also relied heavily on *key-value stores*. The local *key-value store* that was powered by `vmod_kvstore` was featured a lot, but `vmod_redis` was by far the most powerful *key-value store*. The fact that *Redis* is distributed makes it an excellent tool for bridging the gap between the origin and *Varnish* in terms of stateful data.

This example also hinted at other use cases beyond basic web acceleration: there were *API* examples, and examples in which *Varnish* served as an *authentication gateway*. The next chapter is all about alternative domains of application for *Varnish*.

Chapter 9 is the last real chapter of this book and focuses on *Varnish* as *CDN software*. As the internet continues to evolve, and as the need for low latency and high throughput at scale continues to become more important, *Varnish* will become the foundation of your content delivery strategy.

We've come a long way. Let's bring it all together in *chapter 9*.

Chapter 9: Building your own CDN with Varnish

Varnish is most often presented as a *caching proxy* that you put in front of your web servers to protect them from excessive load.

In some cases *Varnish* is installed on the same machine as the web server. In other cases, *Varnish* is installed on one or more separate machines.

In both situations *Varnish* is put as close as possible to the *origin*. While this works well for many websites and other HTTP-based platforms, it is not always the best course of action.

When we talk about *content delivery* and *web acceleration*, our responsibilities are two-fold:

- Platform stability
- Quality of experience for the user

Platform stability is quite straightforward: the caching capabilities of *Varnish* allow it to serve as an *origin shield*.

The quality of experience we strive for, often measured through *latency* and *throughput*, is not only achieved by caching: connectivity also plays a big part in this.

When network latency increases because of network limitations or the geographical location of the user, it makes sense to put a cached version of the content as close to the user as possible. The caching itself will ensure constant throughput at scale. This is what *content delivery networks (CDNs)* are built for, and what this chapter is all about.

In this chapter, we will explain why *Varnish* is excellent *CDN software*, and how you can build your own CDN using both *Varnish Cache* and *Varnish Enterprise*.

9.1 What is a CDN?

In its most basic form, a *content delivery network (CDN)* is nothing more than a bunch of caching nodes.

The reason that it's a *bunch* is related to:

- Storage capacity
- High availability
- Horizontal scalability
- Geographic distribution

CDNs aren't magic, and running them is an effort that combines caching and request routing.

Here's why people use a *CDN*:

- Protect the origin from client requests that cause excessive load
- Reduce infrastructure costs
- Reduce latency by putting cached content close to the user
- Caching large volumes of data

9.1.1 Network connectivity

CDN providers tend to have many *points of presence (PoP)*: these are data center sites where they host a number of caching nodes and where network connectivity is good.

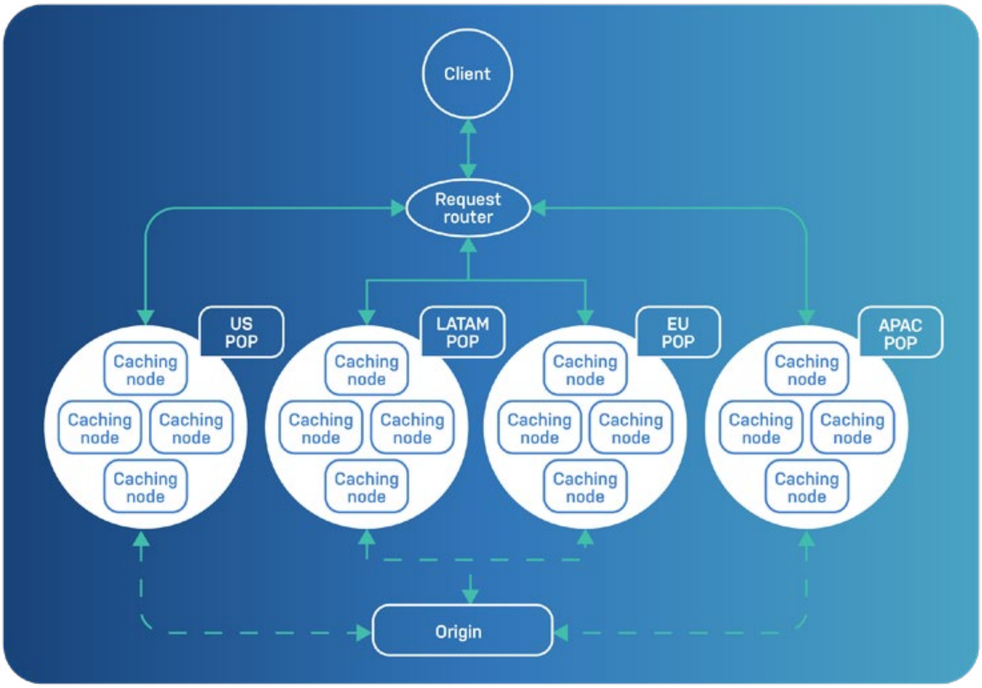
These *PoP*s are typically spread across various geographical locations to ensure network latency is low for as many key regions as possible. Even though *fiber-optic cables* are enormously fast, accessing content that is thousands of miles away from the user can still result in latency.

Having global coverage ensures that any user, regardless of their geographical location, has minimal network latency. In the end, the combination of caching and networking has to result in an acceptable *time to last byte* for any *HTTP resource* that is requested.

Especially for *latency-sensitive* use cases like *OTT video streaming*, having a decent and constant throughput is crucial. And for *live video*, for example in a sports context, any latency seriously impacts the quality of experience.

These *PoP*s are mostly in key geographical areas or areas with significant demand.

Here's a simplified diagram that features four *PoP*s:



CDN diagram

This *CDN* has four *PoP*s:

- A *PoP* in the *United States*
- A *PoP* in *Latin America*
- A *PoP* in the *European Union*
- A *PoP* in the *Asia and Pacific* region

For the sake of simplicity, each *PoP* only has a handful of caching nodes. In reality, *PoP*s can consist of dozens or even hundreds of caching nodes.

9.1.2 Caching

As mentioned earlier, a *CDN* is nothing more than a bunch of caching nodes.

It is the caching that ensures the stability of the *origin platform*. But the fact that a *CDN* has all objects cached globally is a myth: cache storage is precious, and *CDNs* want to be selective about what they cache, in which nodes they want to cache, and for how long.

Quite often it's not about the cache hits; it's about how good your misses are.

It is unrealistic to expect that a *CDN* has enough caching capacity in each *PoP* to cache everything. There is just too much data out there: ranging from *high-resolution images* to *4K-quality video-on-demand catalogs*.

As long as the time it takes to fetch the content from the *origin* is acceptable, there's no real violation of the *Quality Of Experience*. And in the case of *Varnish*, features like content streaming and request coalescing will have a positive impact on both *platform stability* and *Quality Of Experience*.

CDNs also try to figure out how likely it is that anyone else will request the content that is being fetched from the origin. If the content appears to be *long-tail content*, the caching node might decide not to insert the object in cache until it is requested again.

Many *CDN* architectures implement multiple caching tiers, in which each tier has its own role. Some tiers are only there to cache *hot data* and are primarily there to route cache misses to other tiers that have more storage.

Some tiers may operate on a memory-only basis, while other tiers may combine disk storage and memory.

The caching policies of some *CDN providers* might be very complex, depending on their needs.

9.1.3 Request routing

Having caching farms with good network connectivity all over the world is one thing; routing client requests to the right *PoP* is another.

Later in this chapter we will cover some request routing strategies in detail, but at this point we can generalize and say that potential request routing strategies are:

- DNS with geographical awareness
- Services based on HTTP redirection
- Anycast

And quite often it's a mix of various strategies.

The first step involves a basic localization of the client: on which continent is the client located? Does the client IP address match one of the major regions where we have *PoP*s?

The next step may involve network routing methodologies, such as *Anycast*, which announce an IP address in multiple locations and can calculate the shortest route to a *PoP*.

9.1.4 Why build your own CDN?

Although *commercial CDN services* are easy to use, and although they have the scale to cover the most significant parts of the globe, they are black boxes.

For companies that want a tighter grip on their content delivery strategy, relying solely on a *CDN-as-a-service* can prove to be the wrong bet.

At a certain scale, these services can also become expensive. That's why a lot of companies are building their own *CDN*, or at least a selection of *PoP*s that fit into a *hybrid CDN strategy*.

For companies that serve the majority of their traffic from the same geographical region as their origin, it makes sense to build a local *CDN*. Telecom companies and national broadcasters fit into that category. For the latter this is usually related to *OTT video streaming*.

It is also possible that your *CDN* provider doesn't have a *PoP* in an area where a lot of your users are located. This is also a reason why you would build a *Private CDN PoP* there.

What we also see is that companies build a local *CDN* as an *origin shield*: it protects the origin from revalidation requests coming from the various *PoP*s of their *CDN service provider*. The irony is that these revalidation requests are the equivalent of a *DDoS attack*, which requires origin shielding.

Based on these scenarios there are actually three main reasons why companies build their own *CDN*:

- Better coverage
- More control over the *content delivery chain*
- More predictable costs

If you already have data center capacity, networking resources and infrastructure, building your own *CDN* can be a very sensible thing to do.

9.2 Why Varnish?

If you're planning to build your own *CDN*, why should you consider using *Varnish* for the job?

To make our point, we won't present a lot of new information, but instead will reiterate facts we have already mentioned throughout the book.

9.2.1 Request coalescing

Request coalescing ensures that massive amounts of concurrent requests for non-cached content don't cause a stampede of backend requests.

As explained earlier, *request coalescing* will put requests for the same resource on a waiting list, and only send a single request to the origin. The response will be stored in cache and will satisfy all queued sessions in parallel.

In terms of *origin shielding*, this is a killer feature that makes *Varnish* an excellent building block for a *Private CDN*.

9.2.2 Backend request routing

Because of *VCL*, *Varnish* is capable of doing granular routing of incoming requests to the selected backend.

A backend can be the origin server, but it could also be another caching tier that is part of your *CDN strategy*.

`vmod_directors` offers a wide range of load-balancing algorithms, and when content affinity matters, the *shard director* is the director of choice.

Extra logic, written in *VCL*, can even precede the use of directors.

When having to connect to a lot of backends, or connect to backends *on-the-fly*, *Varnish Enterprise's* `vmod_goto` is an essential tool.

9.2.3 Performance and throughput

Varnish is designed for performance and scales incredibly well. If you were to build a *Private CDN* using *Varnish*, the following facts and figures will give you an idea on how it is going to perform.

- An individual *Varnish* server can handle more than *800,000 requests per second*.
- A throughput of over *100 Gbps* can be achieved on a single *Varnish* server where *Hitch* is used to terminate *TLS*.
- *Varnish Enterprise* can handle over *200 Gbps* on a single server using its *native TLS* capabilities.
- In terms of latency, *Varnish* can serve cached objects *sub-millisecond*.

These are not marketing numbers: these numbers were measured in actual environments, both by *Varnish Software* and some of its clients.

Of course you will only attain these numbers if you have the proper hardware, and if your network is fast and stable enough to handle the throughput. Some of the hardware that was used for these benchmarks is incredibly expensive.

In real-world situations on *commercial off-the-shelf hardware*, you will probably not be able to match this performance; however, *Varnish* is still freakishly fast.

9.2.4 Horizontal scalability

It is easy to scale out a cluster of *Varnish* servers to increase the capacity of the *CDN*.

In fact it is quite common to have two layers of *Varnish* for scalability reasons:

- An *edge tier* that stores hot content in memory and routes cache misses to the storage tier via *consistent hashing*
- A *storage tier* that is responsible for storing most of the content catalog

A request routing component selects one of two *edge nodes*. As explained, these edge nodes only contain the most popular objects. Via *consistent hashing* traffic is routed to the storage layer. The sharding director will create a consistent hash and will provide *content affinity*.

This *content affinity*, based on the request URL, will ensure that every miss for a URL on the *edge tier* will also be routed to the same server on the *storage tier*.

Adding storage capacity in your *CDN* is as simple as adding extra *storage nodes*.

Horizontally scaling the *edge tier* is also possible, but the hit rate doesn't matter too much at that level. The only reason to do it is to increase the outward bandwidth of our *PoP*.

Remember the statement earlier in this chapter?

It's not about the cache hits; it's about how good your misses are.

In this case our misses are very good because they are also served by *Varnish* at the *storage-tier level*. There is really no need to beef up your *edge tier* too much as long as it doesn't get crushed under the load of incoming requests.

9.2.1 Transparency

Because of *Varnish*'s unparalleled logging and monitoring tools, the transparency of a *Varnish-based CDN* is quite amazing.

`varnishlog` provides in-depth information about requests, responses, timing, caching, selected backends, and *VCL* decisions. On top of that you can use `std.log()` to log custom messages.

When you start using multiple *Varnish* nodes in a single environment, running `varnishlog` on each node can become tedious. At this point *log centralization* will become important.

Tools like *Logstash* and *Beats* offer plugins for `varnishlog`, which facilitates shipping logs to a central location without having to transfers log files.

In *chapter 7* we already talked about *Prometheus*, and how it has become something of an industry standard for time-series data. `varnishstat` counters can easily be exported and centralized in *Prometheus*. A tool like *Grafana* can be used to visualize these alerts.

In *Varnish Enterprise* `vmod_kvstore` gives you the ability to have custom counters. And on top of that there's *Varnish Custom Statistics*.

Having transparency is important: knowing how your *CDN* is behaving, being able to troubleshoot, and having actionable data to base decisions on. This leads to more control and a better understanding of your *end-to-end delivery*.

Once again *Varnish* proves to be an excellent candidate as *CDN software*.

9.2.1 Varnish Cache or Varnish Enterprise?

It is entirely possible to build your own *CDN* using *Varnish Cache*.

Storage becomes a bit trickier with *Varnish Cache*: we advise against using the *file stevedore*, which means your *storage tier* relies on memory only.

As long as you can equip your storage tier with enough memory and enough nodes, your *CDN* will scale out just fine. Just keep the increased complexity of managing large amounts of servers in mind.

A very important component is the *shard director*. It is responsible for creating the *content affinity* that is required to provide horizontal scalability of your *storage tier*. This director is part `vmod_directors` and is shipped with *Varnish Cache*.

The reality is that the *Massive Storage Engine (MSE)* is a key feature for building a *Private CDN*:

- *MSE* combines the speed of memory and the reliability of disks.
- *MSE* can store petabytes of data on a single machine.
- *MSE* is much more configurable than any other *stevedore*.
- The *Memory Governor* ensures a constant memory footprint on the server.
- *MSE* offers a persisted cache that can survive restarts.
- `vmod_mse` allows *MSE book and store* selection on a per-request basis.

MSE is only available on *Varnish Enterprise* and is the number one reason why people who are building their own *CDN* choose *Varnish Enterprise*.

When your *CDN* increases in size, being able to benefit from the *Varnish Controller* will simplify managing those nodes.

Other than that, choosing between *Varnish Cache* and *Varnish Enterprise* will mainly depend on the *VMODs* you need.

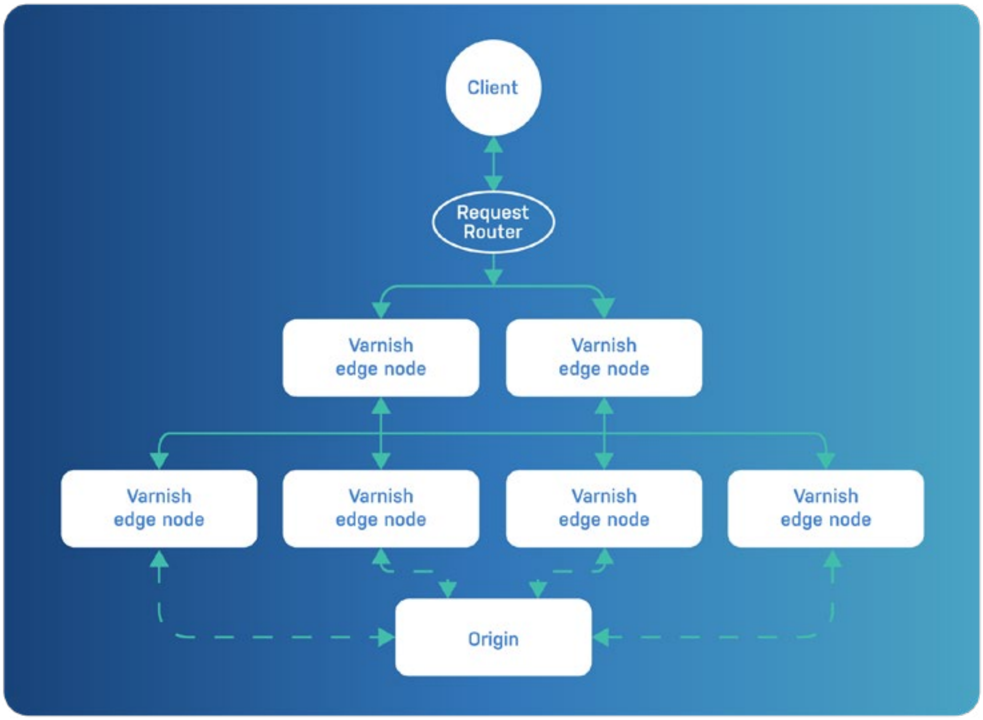
9.3 Varnish CDN architecture

We've mentioned it before: the key objective when building your own *CDN* is *horizontal scalability*. You probably won't be able to serve every single request from one *Varnish node*.

The main reason is not having enough cache storage. Another reason is that one server may not be equipped to handle that many incoming requests.

In essence, *Varnish* receives *HTTP responses* from a backend, which are sent to a client. The backend shouldn't necessarily be the origin, and the client isn't necessarily the end-user. The fact that both the storage and request processing should scale horizontally means that we can use *Varnish* as a building block to develop a *multi-tier architecture*.

The following diagram contains a *multi-tier Varnish environment*. It could be the architecture for a small *PoP*:



Multi-tier Varnish

Let's talk about the various tiers for a moment.

9.3.1 Edge tier

The *edge tier* is responsible for interfacing directly with the clients.

These nodes handle *TLS*. If you're using *Varnish Cache*, *Hitch* is your tool of choice for *TLS termination*. If you're using *Varnish Enterprise*, you can use *Hitch* or *native TLS*.

Any *client-side security precautions*, such as authentication, rate limiting or throttling, are also the responsibility of the *edge tier*.

Any *geographical targeting or blocking* that requires access to the client IP address also happens on the edge.

In terms of *horizontal scalability*, edge nodes are added to enable client-delivery capacity. This means having the bandwidth to deliver all the assets. Because the *edge tier* is directly in contact with the clients, it has to be able to withstand a serious beating and handle all the incoming requests.

Hardware considerations

In terms of hardware, caching nodes in the *edge tier* will need very fast *network interfaces* to provide the desired bandwidth. Other tiers will receive significantly less traffic and need to provide less bandwidth.

Edge nodes also need enough CPU power to handle *TLS*. If your *edge VCL configuration* has compute-intensive logic, having powerful CPUs will be required to deliver the desired bandwidth.

Memory is slightly less important here: the goal is not to serve all objects from cache, but to only serve hot items from cache. Consider that *50%* of the available server memory needs to be allocated via `malloc` or `mse`, and the other *50%* is there for *TCP buffers* and *in-flight content*.

If you're using *MSE's memory governor* feature, you can allocate up to *90%* of your server's memory to `varnishd`.

VCL example

Here's a very basic *VCL example* for an *edge-tier node*:

```
vcl 4.1;

import directors;

backend broadcaster {
    .host = "broadcaster.example.com";
    .port = "8088";
}

backend storage1 {
    .host = "storage1.example.com";
    .port = "80";
}

backend storage2 {
    .host = "storage2.example.com";
    .port = "80";
}

backend storage3 {
    .host = "storage3.example.com";
    .port = "80";
}

acl invalidation {
    "localhost";
    "172.24.0.0"/24;
}

sub vcl_init {
    new storage_tier = directors.shard();
    storage_tier.add_backend(storage1, rampup=5m);
    storage_tier.add_backend(storage2, rampup=5m);
    storage_tier.add_backend(storage3, rampup=5m);
    storage_tier.reconfigure();
}

sub vcl_recv {
    set req.backend_hint = storage_tier.backend(URL);
    if(req.method == "BAN") {
        if (req.http.X-Broadcaster-Ua ~ "^Broadcaster") {
            if (!client.ip ~ invalidation) {
                return(synth(405,"BAN not allowed for " + client.
ip));
            }
            if(!req.http.x-invalidate-pattern) {
                return(purge);
            }
            ban("obj.http.x-url ~ " + req.http.x-invalidate-pattern
                + " && obj.http.x-host == " + req.http.host);
            return (synth(200,"Ban added"));
        } else {
            set req.backend_hint = broadcaster;
            return(pass);
        }
    }
}

sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
    set beresp.http.x-host = bereq.http.host;
    set beresp.ttl = 1h;
}

sub vcl_deliver {
    set resp.http.x-edge-server = server.hostname;
    unset resp.http.x-url;
    unset resp.http.x-host;
}
```

The only *enterprisy* part of this *VCL* is the *broadcaster* implementation: when a `BAN` request is received by the *edge tier*, and the `X-Broadcaster-Ua` header doesn't contain `Broadcaster`, we connect to the *broadcaster* endpoint and let it handle invalidation on all selected nodes.

If the `X-Broadcaster-Ua` request header does contain `Broadcaster`, it means it's the *broadcaster* connecting to the node, and we handle the actual *ban*.

Apart from that, this example is compatible with *Varnish Cache*.

As you can see, the *shard director* front and center in this example because it is responsible for distributing requests to the *storage tier*. A hash key is composed, based on the URL, for every request. The *sharding director* is responsible for mapping that hash key to a backend on a consistent basis.

This means every cache miss for a URL is routed to the same storage server. If the hit rate on certain objects is quite low, but the request rate is very high, there is a risk that the selected storage node becomes overwhelmed with requests. This is something to keep an eye on from an operational perspective.

You can throw in as much logic on the *edge tier* as you want, depending on the *VMODs* that are available to you. We won't go into detail now, but in the previous chapters there were plenty of examples. Specifically in *chapter 8*, which is all about *decision-making on the edge*, you'll find plenty of inspiration.

9.3.2 Storage tier

Because of *content affinity*, our main priority is to achieve a much higher hit rate on the *storage tier*.

Every node will cache a *shard* of the total cached catalog. We use the word *shard* on purpose because the *shard director* on the *edge-tier* level will be responsible for routing traffic to *storage-tier nodes* using a *consistent hashing algorithm*.

Hardware considerations

If you're using *Varnish Cache*, having enough memory is your main priority: as long as the assigned memory as a total sum of *storage nodes* matches the catalog of resources, things will work out and your hit rate will be good.

If you're using *Varnish Enterprise*, the use of *MSE* as your stevedore is a no-brainer: assign enough memory to store the *hot data* in memory, and let *MSE's persistent storage* handle the rest. We advise using *NVMe SSD disks* for persistence to ensure that disk access is fast enough to serve long-tail content without too much latency.

We also advise that you set *MSE*'s `memcache_size` configuration setting to `auto`, which enables the *memory governor* feature. By default *80%* of the server's memory will be used by `varnishd`.

CPU power and very fast network interfaces aren't a priority on the *storage tier*: most requests will be handled by the *edge tier*. Only requests for long-tail content should end up being requested on the *storage tier*.

VCL example

The *VCL example* for the *storage tier* focuses on the following elements:

- Banning objects
- Providing `stale-if-error` support
- Choosing the right *MSE store*
- Protecting the origin from malicious requests by enabling the *WAF*

Here's the code:

```
vcl 4.1;

include "waf.vcl";

import stale;
import mse;

acl invalidation {
    "localhost";
    "172.24.0.0"/24;
    "172.18.0.0"/24;
}

sub vcl_init {
    varnish_waf.add_files("/etc/varnish/modsec/modsecurity.conf");
    varnish_waf.add_files("/etc/varnish/modsec/owasp-crs-v3.1.1/
crs-setup.conf");
    varnish_waf.add_files("/etc/varnish/modsec/owasp-crs-v3.1.1/
rules/*.conf");
}

sub vcl_recv {
    if(req.method == "BAN") {
        if (!client.ip ~ invalidation) {
            return(synth(405,"BAN not allowed for " + client.ip));
        }
        if(!req.http.x-invalidate-pattern) {
            return(purge);
        }
        ban("obj.http.x-url ~ " + req.http.x-invalidate-pattern
            + " && obj.http.x-host == " + req.http.host);
        return (synth(200,"Ban added"));
    }
}

sub stale_if_error {
    set beresp.keep = 1d;
    if (beresp.status >= 500 && stale.exists()) {
        stale.revive(20m, 1h);
        stale.deliver();
        return (abandon);
    }
}

sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
    set beresp.http.x-host = bereq.http.host;
    call stale_if_error;
    if (beresp.ttl < 120s) {
        mse.set_stores("none");
    } else {
        if (beresp.http.Content-Type ~ "^video/") {
            mse.set_stores("store1");
        } else {
            mse.set_stores("store2");
        }
    }
}

sub vcl_backend_error {
    call stale_if_error;
}

sub vcl_deliver {
    set resp.http.x-storage-server = server.hostname;
    unset resp.http.x-url;
    unset resp.http.x-host;
}
```

When we receive `BAN` requests, we ensure the necessary logic is in place to process them and to prevent unauthorized access.

Via a custom `stale_if_error` subroutine, we also provide a safety net in case the origin goes down: by setting `beresp.keep` to a day, expired and out-of-grace objects will be kept around for a full day.

When the *origin* cannot be reached, `vmod_stale` will *revive* objects, make them fresh for another *20 minutes*, and give them *an hour of grace*. The object revival only takes place when the object is available and if the origin starts returning *HTTP 500*-range responses.

This *VCL example* also has *WAF* support. The *WAF* is purposely placed in the *storage tier* and not in the *edge tier*.

Depending on the number of *WAF rules*, their complexity, and the amount of traffic your *Varnish CDN* is processing, the *WAF* can cause quite a bit of overhead. We want to place it in the tier that receives the fewest requests to reduce this overhead.

Another reason why the *WAF* belongs as close to the origin as possible is because our goal is to protect the origin from malicious requests, not necessarily *Varnish* itself. And we also want to avoid that bad requests end up in the cache.

And finally `vmod_mse` is used to select *MSE stores*: in this case `store1` is used to store video footage, and `store2` is used for any other content.

9.3.3 Origin-shield tier

There is an implicit tier that also deserves a mention: *the origin-shield tier*.

When one of the *CDN's PoPs* is in the same data center as the *origin server*, the *storage tier* will assume the role of *origin-shield tier*.

When your *CDN* has many *PoPs*, none of which are hosted in the same data center as the *origin server*, it makes sense to build a *small, local CDN* that protects the origin from the side effects of incoming requests from the *PoPs*.

Even if you're not planning to build your own *CDN*, and you rely on *public CDN providers*, it still makes sense to build a *local CDN*, especially if the *origin server* is prone to heavy load. This way *CDN cache misses* will not affect the stability of the *origin server*.

Typical tasks that the *origin-shield tier* will perform are:

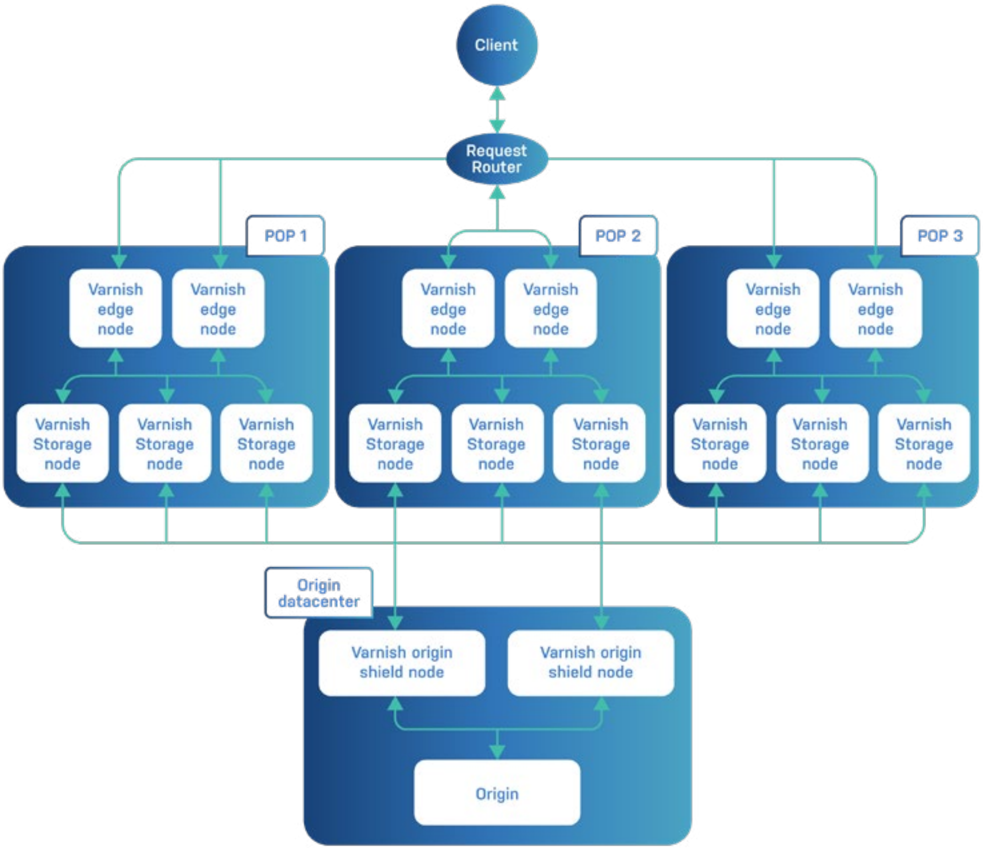
- Defining caching rules
- `stale-if-error` behavior
- *WAF* protection
- Using `vmod_directors` to route requests to the right *origin server*

We won't present a dedicated *VCL example* for the *origin-shield* tier because the code will be nearly identical to the one presented in the *storage tier*. Only the `vmod_mse` will not be part of the *VCL code*.

This is the tier that receives the fewest requests. The hardware required for this tier should only be able to handle requests coming from the *storage tier*, which in its turn only receives requests that weren't served by the *edge tier*.

If you have a dedicated *origin-shield tier*, this is also the place where the *WAF* belongs: close to the origin and in a tier that receives the fewest requests.

Instead of a *VCL example*, here's a diagram that includes three *PoPs*, each with *two tiers*, and a dedicated *origin-shield tier* in the *origin data center*:



Varnish CDN with a dedicated origin-shield tier

Here's the scenario for this diagram:

- The request router will pick an *edge node* in the selected *PoP*.
- When the object cannot be served by the *edge tier*, the fetch is done from the *storage tier*.
- Because of *content affinity*, the same *storage node* will always be selected when a specific URL is requested.
- When the *storage node* cannot serve the object from cache, an *origin-shield node* is selected.
- When the *origin-shield node* cannot serve the object from cache, the *origin server* receives the request.

9.4 Caching video

As mentioned in *chapter 1*: the majority of the internet's bandwidth is no longer consumed by websites or images: today online video is responsible for more than 80% of internet traffic.

This type of video streaming is called *over-the-top (OTT) video*. This means that traditional cable, broadcast and satellite platforms are bypassed and the internet is used instead.

OTT video is served over *HTTP*, which makes *Varnish* an excellent candidate to cache and deliver it to clients.

We also highlighted this fact in *chapter 1*: a single *4K video stream* consumes at least *6 GB per hour*. As a result, hosting a large catalog of *on-demand video* will require lots of storage. Combined with potential viewers spread across multiple geographic regions, delivering *OTT video* through a *CDN* is advised.

9.4.1 OTT protocols

First things first: there are various *OTT protocols*, we're going to highlight three of them:

- HLS
- MPEG-DASH
- CMAF
- These protocols have some similarities:
- The video stream is fragmented into a sequence video segments.
- Each video segment represents a number of seconds of footage.
- A manifest file, also called a *playlist* file, indexes the various segments per bitrate and provides metadata.
- The video player reads the manifest file and loads the corresponding sequence of video segments.
- From capturing to viewing footage, there are a couple of steps that take place:
- Captured footage is sent to an *encoder* or a *transcoder* to convert it using a supported *codec*.
- Encoded footage is then sent to an ingestion service, which chops the footage up into various segments.
- The segmented footage, along with the playlist, is then published on the *content delivery platform*, which is consumed by video players.

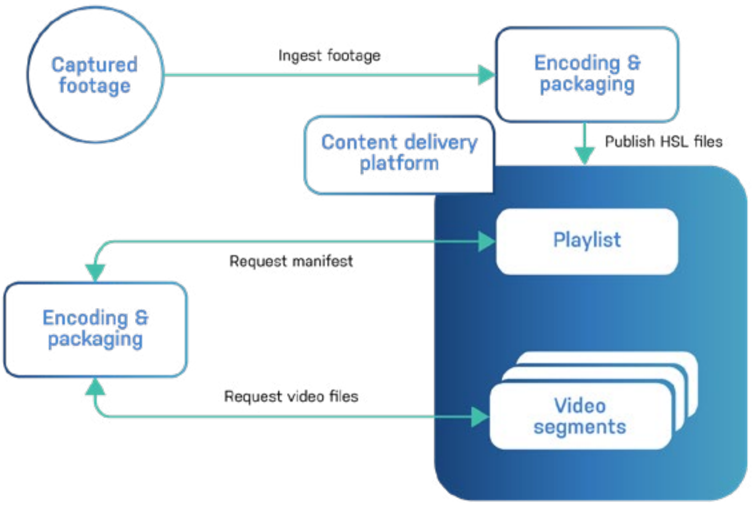
This process is also called content packaging.

For *live streaming*, the footage comes in as it is captured. There is a slight delay: footage is only sent to the player when the video segment is completed. When each video segment represents six seconds of footage, the delay is around six seconds.

The corresponding manifest file is updated each time a new segment is added. It's up to the video player to refresh the manifest file and load the newest video segment.

When you add *catch-up capabilities* to live streaming, a viewer can pause the footage, rewind and fast-forward. This requires older video segments to remain available in the manifest file.

Video on demand (VoD) has no live component to it, which results in a fixed manifest file. Because all video segments are ready at the time of viewing, the video player should only load the manifest file once.



OTT video streaming flow

HLS

HTTP Live Stream (HLS) is a very popular streaming protocol that was invented by *Apple*.

Video footage is encoded using the `H.264/AVC` or `HEVC/H.265` codec. The container format for the segmented output is either `fMP4` *(fragmented MP4)*, or `MPEG-TS` *(MPEG transport streams)*.

HLS supports various *resolutions* and *frame rates*, which impact the *bitrate* of the stream.

Resolutions range from `256x144` for *144p* all the way up to `3840x2160` for *4K*, while frame rates range from *23.97 fps* to *60 fps*. The resulting bitrates are usually between *300 Kbps* and *50 Mbps*.

The endpoints of the video segments are listed in an *extended M3U manifest* file. Usually, this file has a `.m3u8` extension and acts as a playlist.

The manifest file is consumed by the video player, and the player loads the video segments in the order in which they were listed in the `.m3u8` file.

On average, every video segment represents *between six and ten seconds* of footage. The target duration of the video segments is defined in the manifest file.

Audio is encoded in `AAC`, `MP3`, `AC-3` or `EC-3` format. The audio is either part of the video files or is listed in the manifest file as separate streams in case of multi-language audio.

Endpoints referring to *captions* and *subtitles* can also be added to the `.m3u8` file.

HLS supports *adaptive bitrate streaming*. This means that the *HLS manifest file* offers multiple streams in different resolutions based on the available bitrate. It's up to the video player to detect this and select a lower or higher bitrate based on the available bandwidth.

Because the video stream is chopped up in segments, *adaptive bitrate streaming* will allow a change in quality after every segment.

Video segments can also be encrypted using *AES encryption* based on a *secret key*. This key is mentioned in the manifest, but access to its endpoint should be protected to avoid unauthorized access to the video streams.

In most cases securing a video stream usually happens via a *third-party DRM provider*. This is more secure than having the *secret key* out in the open or relying on your own authentication layer to protect that key.

DRM stands for *digital rights management* and does more than just provide the key to decrypt the video footage. The *DRM service* also contains rules that decide whether or not the user can play the video.

This results in a number of extra playback features:

- Allow or block offline playback
- Restrict access to specific platforms
- Restrict a user from watching a video more than *x* times
- Decide when the user can watch the video
- Restrict how long the video remains available

Here's an example of a *VoD manifest file*:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:6
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-PLAYLIST-TYPE:VOD
#EXTINF:6.000000,
stream_00.ts
#EXTINF:6.000000,
stream_01.ts
#EXTINF:6.000000,
stream_02.ts
#EXTINF:6.000000,
stream_03.ts
#EXTINF:6.000000,
stream_04.ts
#EXTINF:6.000000,
stream_05.ts
#EXTINF:6.000000,
stream_06.ts
#EXTINF:6.000000,
stream_07.ts
#EXTINF:6.000000,
stream_08.ts
#EXTINF:6.000000,
stream_09.ts
#EXTINF:5.280000,
stream_010.ts
#EXT-X-ENDLIST
```

This `.m3u8` file refers to a video stream that is segmented into *11 parts*, where the duration of every segment is *six seconds*. The last segment only lasts *5.28* seconds.

If the manifest file is hosted on `https://example.com/vod/stream.m3u8`, the video segments will be loaded by the video player using the following URLs:

```
https://example.com/vod/stream_00.ts
https://example.com/vod/stream_01.ts
https://example.com/vod/stream_02.ts
https://example.com/vod/stream_03.ts
https://example.com/vod/stream_04.ts
https://example.com/vod/stream_05.ts
https://example.com/vod/stream_06.ts
https://example.com/vod/stream_07.ts
https://example.com/vod/stream_08.ts
https://example.com/vod/stream_09.ts
https://example.com/vod/stream_010.ts
```

The `.m3u8` manifest supports a lot more syntax, which is beyond the scope of this book. We will conclude with an example of *adaptive bitrate streaming*:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-STREAM-INF:BANDWIDTH=800000,RESOLUTION=640x360
stream_360p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=1400000,RESOLUTION=842x480
stream_480p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2800000,RESOLUTION=1280x720
stream_720p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=5000000,RESOLUTION=1920x1080
stream_1080p.m3u8
```

This manifest file provides additional information about the stream and loads other manifest files based on the available bandwidth. Each of these `.m3u8` files links to the video segments that contain the required footage.

MPEG-DASH

The *DASH* part of *MPEG-DASH* is short for *Dynamic Adaptive Streaming over HTTP*. It is quite similar to and is newer than *HLS*.

MPEG-DASH was developed as an official standard at a time when *Apple's HLS protocol* was competing with protocols from various other vendors. *MPEG-DASH* is now an open source standard that is pretty much on par with *HLS* in terms of capabilities.

As an open standard, *MPEG-DASH* used to have a slight edge in terms of capabilities, but *HLS* matched these with later upgrades to the protocol.

The main difference is that *MPEG-DASH* is *codec-agnostic*, whereas *HLS* relies on `H.264/AVC` and `HEVC/H.265`.

MPEG-DASH streams are not supported by *Apple devices* because they only use *HLS*.

The manifest file for *MPEG-DASH* streams is in *XML format*. This is a bit harder to interpret, but it allows for richer semantics and more context.

Here's an example:

```
<?xml version="1.0" ?>
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" profiles="urn:mpeg:dash:-
profile:isoff-live:2011" minBufferTime="PT2.00S" mediaPresentationDu-
ration="PT2M17.680S" type="static">
  <Period>
    <!-- Video -->
    <AdaptationSet mimeType="video/mp4" segmentAlignment="true"
startWithSAP="1" maxWidth="1920" maxHeight="1080">
      <SegmentTemplate timescale="1000" duration="1995" initial-
ization="$RepresentationID$/init.mp4" media="$RepresentationID$/
seg-$Number$.m4s" startNumber="1"/>
      <Representation id="video/avc1" codecs="avc1.640028"
width="1920" height="1080" scanType="progressive" frameRate="25"
bandwidth="27795013"/>
    </AdaptationSet>
    <!-- Audio -->
    <AdaptationSet mimeType="audio/mp4" startWithSAP="1" segmen-
tAlignment="true" lang="en">
      <SegmentTemplate timescale="1000" duration="1995" initial-
ization="$RepresentationID$/init.mp4" media="$RepresentationID$/
seg-$Number$.m4s" startNumber="1"/>
      <Representation id="audio/en/mp4a" codecs="mp4a.40.2" band-
width="132321" audioSamplingRate="48000">
        <AudioChannelConfiguration schemeIdUri="urn:mpeg:-
dash:23003:3:audio_channel_configuration:2011" value="2"/>
      </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

This manifest file, which could be named `stream.mpd`, uses an `<MPD></MPD>` tag to indicate that this is an *MPEG-DASH* stream. The `minBufferTime` attribute tells the player that it can build up a *two-second buffer*. The `mediaPresentationDuration` attribute announces that the stream has a duration of *two minutes and 17.68 seconds*. The `type="static"` attribute tells us that this a *VoD stream*.

Within the manifest, there can be multiple *consecutive periods*. However, this example only has one `<Period></Period>` tag.

Within the `<Period></Period>` tag we defined multiple `<AdaptationSet></AdaptationSet>` tags. An *adaptation set* is used to present audio or video in a specific bitrate. In this example the delivery of audio and video comes from different files, hence the two adaptation sets.

An *adaptation set* has two underlying entities:

- A *segment template* that defines where the corresponding files can be found and when they should be loaded
- A *representation* that defines the properties of the audio and video segments that are loaded

In this example the video footage can be found in the `video/avc1` folder. This location is referenced in the `$RepresentationID$` variable, which is then used within the `<SegmentTemplate></SegmentTemplate>` tag.

The `duration` attribute defines the duration of each audio or video segment. In this case this is `1995` milliseconds.

Another variable is the `$Number$` variable, which in this case starts at `1` because of the `startNumber="1"` attribute.

The maximum value of `$Number$` can be calculated by dividing the `mediaPresentationDuration` attribute in the `<MPD></MPD>` tag by the `duration` attribute in the `<SegmentTemplate></SegmentTemplate>` tag:

```
137680 / 1995 = 69
```

The total duration of the stream is `137680` milliseconds, and each segment represents `1995` milliseconds of footage. This results in *69 audio and video segments* for this stream.

The `$RepresentationID$/seg-$Number$.m4s` notation results in the following video segments:

```
video/avc1/seg-1.m4s
video/avc1/seg-2.m4s
video/avc1/seg-3.m4s
...
video/avc1/seg-69.m4s
```

The `<Representation></Representation>` tag has an `id` attribute that refers to the folder where this representation of the footage can be found. In this case this is `video/avc1`. The `codec` attribute refers to the codec that was used to encode the footage. This is one of the main benefits of *MPEG-DASH*: begin *coded-agnostic*.

The `width` and `height` attributes define the *aspect ratio* of the video, and the `frameRate` attribute defines the corresponding frame rate. And finally the `bandwidth` attribute defines the bandwidth that is required to play the footage fluently. This is in fact the *bitrate* of the footage.

The audio has a similar *adaptation set*.

And based on the `<SegmentTemplate></SegmentTemplate>` and `<Representation></Representation>` tags, you'll find the following audio segments:

```
audio/en/mp4a/seg-1.m4s
audio/en/mp4a/seg-2.m4s
audio/en/mp4a/seg-3.m4s
...
audio/en/mp4a/seg-69.m4s
```

As you can see, the *MPD XML format* caters to multiple audio tracks.

In terms of HTTP, these *MPEG-DASH* resources can be loaded through the following endpoints:

```
https://example.com/vod/stream.mpd
https://example.com/vod/video/avc1/init.mp4
https://example.com/vod/video/avc1/seg-1.m4s
https://example.com/vod/video/avc1/seg-2.m4s
https://example.com/vod/video/avc1/seg-3.m4s
...
https://example.com/vod/audio/en/mp4a/init.mp4
https://example.com/vod/audio/en/mp4a/seg-1.m4s
https://example.com/vod/audio/en/mp4a/seg-2.m4s
https://example.com/vod/audio/en/mp4a/seg-3.m4s
...
```

CMAF

The *Common Media Application Format (CMAF)* is a specification that uses a single encoding and packaging format, yet presents the segmented footage via various manifest types.

Whereas *HLS* primarily uses `MPEG-TS` for its file containers, *MPEG-DASH* primarily uses `fMP4`. When you want to offer both *HLS* and *MPEG-DASH* to users, you need to encode the same audio and video twice.

This leads to a lot of overhead in terms of packaging, storage and delivery.

CMAF does not compete with *HLS* or *MPEG-DASH*. The specification aims to create a uniform standard for segmented audio and video that can be used by both *HLS* and *MPEG-DASH*.

The output will be very similar to the *MPEG-DASH* example:

- There will be separate `seg-*.m4s` files for the audio and video streams.
- There will be `init.mp4` files to initialize audio and video streams.
- There will be a `stream.mpd` file that exposes the footage as *MPEG-DASH*.
- And there will also be a `stream.m3u8` file that exposes the footage as *HLS*.
- The `stream.m3u8` file will refer to various other *HLS manifests* for audio and video for each available bitrate.

9.4.2 Varnish and video

Now that you know about *HLS, MPEG-DASH and CMAF*, we can focus on *Varnish* again.

Because these *OTT protocols* leverage the *HTTP* protocol for transport, *Varnish* can be used to cache them.

We've already mentioned that the files can be rather big: a *4K stream* consumes at least *6 GB per hour*. The *MPEG-DASH* example above has a duration of *two minutes and 17 seconds* and consumes *330 MB*. Keep in mind that this is only for a single bitrate. If you support multiple bitrates, the size of the streams increases even more.

A single *Varnish* server will not cut it for content like this: for a large video catalog, you probably won't have enough storage capacity on a single machine. And more importantly: if you start serving video at scale, you're going to need enough computing resources to handle the requests and data transfer.

The obvious conclusion is that a *CDN* is required to serve *OTT video streams* at scale, which is the topic of this chapter.

Earlier in the chapter we stated that *Varnish Cache* can be used to build your own *CDN*, and we still stand by it. However, if you are serving terabytes of video content, your *CDN's storage tier* will need *Varnish Cache* servers with a lot of memory, and potentially a lot of servers to scale this tier.

However, in this case *Varnish Enterprise's MSE stevedore* really shines. With *MSE* in your arsenal, your storage tier isn't going to be that big. The memory consumption will primarily depend on how popular certain video content is. The *memory governor* will ensure a constant memory footprint on your system, whereas the *persistence layer* will cache the rest of the content.

The *time to live* you are going to assign to *OTT video streams* is pretty straightforward:

- Segmented video files for *HLS* and *MPEG-DASH* can be cached for a very long time.
- Manifest files like `.m3u8` and `.mpd` for *video on demand (VoD)* footage can also be cached for a long time.
- Manifest files like `.m3u8` and `.mpd` for *live streams* should only be cached for half of the duration of a video segment.

Once a video segment is created, it won't change any more. Even for live streaming, you will just add segments, you won't be changing them. A *TTL* of one day or longer is perfectly viable.

The same applies to the manifest files for *VoD*: all video segments are there when playback starts, which means that the manifest file will not change either. Long *TTLs* are fine in this case.

However, for live streaming, caching manifest files for too long can become problematic. A live stream will constantly add new video segments, which should be referenced in the manifest file. This means the manifest file is updated every time.

The update frequency depends on the *target duration* of the segment. If a video segment contains *six seconds* of footage, caching the manifest file for longer than *six seconds* will prevent smooth playback of the footage. The rule of thumb is to only cache a manifest file for half of the target duration.

If your *HLS or MPEG-DASH* stream has a target duration of six seconds, setting the *TTL* for `.m3u8` and `.mpd` files to *three seconds* is the way to go.

Keep in mind that three seconds is less than the `shortlived` runtime parameter, which has a default value of *ten seconds*. This means that these objects don't end up in the regular cache, but in *transient storage*. Don't forget that *transient storage* is unbounded by default, which may impact the stability of your system.

Even when using standard *MSE* instead of standard `malloc`, short-lived objects will end up in an unbounded `Transient` stevedore. It is possible to limit the size of *transient storage*, but that might lead to errors when it is full.

The most reliable way to deal with short-lived content, like these manifest files, is to use the *memory governor*: the memory governor will shrink and grow the size of *transient storage* based on the memory consumption of the `varnishd` process.

What is also unique to the *memory governor* is that it introduces an *LRU mechanism* on transient objects. This ensures that when transient is full, *LRU* takes place rather than returning an error because the transient storage is full.

In reality it seems unlikely that *short-lived objects containing the manifest files* would be the reason that *transient storage* spins out of control and causes your servers to go out of memory. Because we're dealing with relatively small plain text files, the size of each manifest will be mere kilobytes.

9.4.3 VCL tricks

There are some interesting things we can do with *VCL* with regard to video. These are individual examples that only focus on video. Of course these *VCL snippets* should be part of the *VCL code* that is in one of your *CDN tiers*.

Controlling the TTL

The first example is pretty basic, and ensures that `.m3u8`, `.mpd`, `.ts`, `.mp4` and `.m4s` files are always cached. Potential cookies are stripped, and the *TTL* is tightly controlled:

```
vcl 4.1;

sub vcl_recv {
   if(req.url ~ "^[^?]*\.(m3u8|mpd|ts|mp4|m4s)(\?.*)?$") {
       unset req.http.Cookie;
       return(hash);
   }
}

sub vcl_backend_response {
   if(bereq.url ~ "^[^?]*\.(m3u8|mpd|ts|mp4|m4s)(\?.*)?$") {
       unset beresp.http.Set-Cookie;
       set beresp.ttl = 1d;
   }

   if(bereq.url ~ "^/live/[^?]*\.(m3u8|mpd)(\?.*)?$") {
       set beresp.ttl = 3s;
   }
}
```

All video-related files are stored in cache for a full day. But if an `.m3u8` or `.mpd` manifest file is loaded where the URL starts with `/live`, it implies this is a live stream. In that case the *TTL* is reduced to half the duration of a video segment. In this case the *TTL* becomes *three seconds*.

Prefetching segments

When dealing with *VoD streams*, we know that all the video segments are ready to be consumed when playback starts, unlike *live streams* where new segments are constantly added.

This allows *Varnish* to prefetch the next video segment, knowing that it is available and is about to be required by the video player. Having the next segment ready in cache may reduce latency at playback time.

Here's the code:

```
vcl 4.1;

import http;

sub vcl_recv {
    if(req.url ~ "^[^?]*\.(m3u8|mpd|ts|mp4|m4s)(\?.*)?$") {
        unset req.http.Cookie;
        if(req.url ~ "^/vod/[^?]*\.(ts|mp4|m4s)(\?.*)?$") {
            http.init(0);
            http.req_copy_headers(0);
            http.req_set_method(0, "HEAD");
            http.req_set_url(0, http.prefetch_next_url());
            http.req_send_and_finish(0);
        }
        return(hash);
    }
}

sub vcl_backend_response {
    if(bereq.url ~ "^[^?]*\.(m3u8|mpd|ts|mp4|m4s)(\?.*)?$") {
        unset beresp.http.Set-Cookie;
        set beresp.ttl = 1d;
    }

    if(bereq.url ~ "^/live/[^?]*\.(m3u8|mpd)(\?.*)?$") {
        set beresp.ttl = 3s;
    }
}
```

This *prefetching* code will fire off an internal subrequest while not waiting for the response to come back. We assume that the next segment will be stored in cache by the time it gets requested.

`http.prefetch_next_url()` does some guesswork on what the next segment's sequence number would be. If a request for `/vod/stream_01.ts` is received, `http.prefetch_next_url()` will return `/vod/stream_02.ts` as the next URL.

However, if the URL would be `/vod/video1/stream_01.ts`, there are two numbers in the URL, which may trigger `http.prefetch_next_url()` to only increase the first number. To avoid this, we can add a prefix, which instructs `http.prefetch_next_url()` to only start increasing numbers after the prefix pattern has been found.

With `/vod/video1/stream_01.ts` in mind, this is what the prefetch function would look like:

```
http.prefetch_next_url("^/vod/video1/");
```

No origin

As mentioned in *chapter 5*: `vmod_file` can serve files from the local file system, and expose itself as a *file backend*.

This could be a useful feature that eliminates the need for an *origin web server*. Here's the *VCL* code:

```
vcl 4.1;

import file;

backend default {
    .host = "origin.example.com";
    .port = "80";
}

sub vcl_init {
    new fs = file.init("/var/www/html/");
}

sub vcl_backend_fetch {
    if(bereq.url ~ "^[^?]*\.(m3u8|mpd|ts|mp4|m4s)(\?.*)?$") {
        set bereq.backend = fs.backend();
    } else {
        set bereq.backend = default;
    }
}
```

The standard behavior is to serve content from the `origin.example.com` backend. This could be the web application that relies on a database to visualize dynamic content.

But the video files are static, and they can be served directly from the file system when *Varnish* has them on disk. This example will match extensions like `.m3u8`, `.mpd`, `.ts`, `.mp4` and `.m4s` and serve these directly from the file system before storing them in the cache.

Ad injection

Imagine the following `.m3u8` playlist:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:6
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-PLAYLIST-TYPE:VOD
#EXTINF:6.000000,
stream_00_us.ts
#EXTINF:6.000000,
stream_01.ts
#EXTINF:6.000000,
stream_02.ts
#EXTINF:6.000000,
stream_03.ts
#EXTINF:6.000000,
stream_04.ts
#EXTINF:6.000000,
stream_05.ts
#EXTINF:6.000000,
stream_06.ts
#EXTINF:6.000000,
stream_07.ts
#EXTINF:6.000000,
stream_08.ts
#EXTINF:6.000000,
stream_09.ts
#EXTINF:5.280000,
stream_010.ts
#EXT-X-ENDLIST
```

The first segment that is loaded is `stream_00_us.ts`, which has `us` in the filename. That is because it is a *pre-roll ad* that is valid for the US market.

Via *geolocation* you can determine the user's location. This is based on the client IP address. That's pretty straightforward. But having to create a separate `.m3u8` file per country is not ideal.

We can leverage `vmod_edgestash` and template this value. Here's what this would look like:

```
stream_00_{{country}}.ts
```

And now it's just a matter of parsing in the right country code. Here's the *VCL* to do that:

```
vcl 4.1;

import edgestash;
import mmdb;

sub vcl_init {
    new geodb = mmdb.init("/path/to/db");
}

sub vcl_recv {
     set req.http.x-country = geodb.country_code(client.ip);
    if (req.http.x-country !~ "^(gb|de|fr|nl|be|us|ca|br)$") {
        set req.http.x-country = "us";
    }
}

sub vcl_backend_response {
    if (bereq.url ~ "\.m3u8$") {
        edgestash.parse_response();
        set beresp.ttl = 3s;
    }
}

sub vcl_deliver {
    if (edgestash.is_edgestash()) {
        edgestash.add_json({"
            {
                "country": ""} + req.http.x-country + {""
            }
        "});
        edgestash.execute();
    }
}
```

This example has ad-insertion capabilities for: the UK, Germany, the Netherlands, Belgium, the US, Canada and Brazil. Users visiting from any other country will see the US pre-roll ad.

9.5 Request routing

You can build a *CDN* with as many *PoP*s as you want, but you need a way to route clients to the right *PoP*.

There are various ways you can do this, but *DNS* is a popular one. The *authoritative DNS server* for the hostname that receives the *DNS requests* uses *geoIP lookups* based on the client network address in the case of *EDNS*, or in non-*EDNS* cases we use the recursive resolver's IP address, which returns the IP address of the *PoP*.

Resolving nameservers can pass information about users using *EDNS Client Subnet*. The subnet is a short suffix that is appended to the end of an IP address that indicates where a user is located. Not all resolvers forward this information.

Another way is via *WAN Anycast*, where the network routing technology is used to select a *PoP* based on the shortest network route.

Certain use cases can also warrant the use of *HTTP* for routing requests to the right *PoP*: a discovery service can also use *geoIP* to localize the client, and then perform an *HTTP 301* redirect to the right *PoP*. Using *HTTP* has the upside that the discovery service will see the requesting client IP and the *geoIP* filtering will be more fine-grained.

It is possible to combine these methods and first perform a crude localization through *DNS*, and then let *Anycast* find the closest server for that IP address.

In a more practical sense, we will cover four request routing implementations:

- PowerDNS
- AWS Route 53
- Anycast
- Varnish Traffic Router

9.5.1 PowerDNS

PowerDNS is an open source DNS server that is quite easy to install and manage.

Via its `geoip` backend plugin, geolocation can be performed. If the request is done with *EDNS* the client network address is part of the DNS request. If it is done without *EDNS* the requesting resolver address is inspected and matched to a *geoIP database*. The DNS response is the IP address of a *PoP* in our *CDN*.

In terms of configuration, you can add the following settings to `/etc/powerdns/pdns.conf`:

```
launch=geoip
geoip-database-files=/usr/share/GeoIP/GeoIP.dat,/usr/share/GeoIP/
GeoIPv6.dat
geoip-zones-file=/etc/powerdns/zone
```

This configuration enables the `geoip` backend, loads the *geoIP* databases, and sets the location of the *zone file*.

The zone file, located in `/etc/powerdns/zone`, contains information about the domain and its records and could look like this:

```
- domain: geo.example.com
  ttl: 60
  records:
    geo.example.com:
       - soa: ns.example.com. hostmaster.example.com. 1 7200 3600
86400 60
       - ns:  ns.example.com.
    eu.geo.example.com:
       - a: 192.168.1.2
    na.geo.example.com:
       - a: 192.168.1.3
    sa.geo.example.com:
       - a: 192.168.1.4
    af.geo.example.com:
       - a: 192.168.1.5
    as.geo.example.com:
       - a: 192.168.1.6
    "*.geo.example.com":
       - a: 192.168.1.7
  services:
     www.geo.example.com: '%cn.geo.example.com'
```

This zone file provides DNS information for the `geo.example.com` domain. There are a certain number of address records available that are linked to specific IP addresses:

- `eu.geo.example.com` points to IP address `192.168.1.2` and represents our *European PoP*.
- `na.geo.example.com` points to IP address `192.168.1.3` and represents our *North American PoP*.
- `sa.geo.example.com` points to IP address `192.168.1.4` and represents our *South American PoP*.

- `af.geo.example.com` points to IP address `192.168.1.5` and represents our *African PoP*.
- `as.geo.example.com` points to IP address `192.168.1.6` and represents our *Asian PoP*.
- `*.geo.example.com` points to IP address `192.168.1.7` and catches DNS requests for unmatched continents, or when the continent information could not be retrieved from the client IP address.

And finally, there is a service definition for `www.geo.example.com` that is exposed as a `CNAME` record. It points to `%cn.geo.example.com`. The `%cn` placeholder is replaced with the continent code of the client.

For any matching address record, the IP address will be returned. Unmatched address records will be caught by the `*.geo.example.com` record.

Because DNS resolution is distributed, it scales well: your system's *DNS resolvers* will perform all the heavy lifting. DNS requests to our *PowerDNS* server will only be made if the cached value of your DNS resolver expires.

As you can see, DNS uses caching techniques just like *Varnish*. There is also a *TTL* that should be respected. However, there is no way to enforce this *TTL*, and no way to forcefully invalidate the cache.

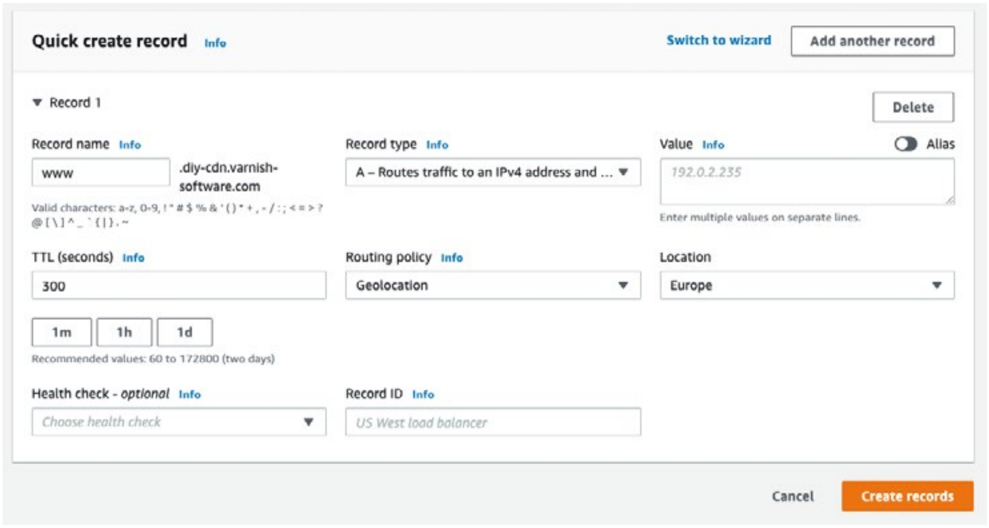
If changes in the zone file occur, it could take a couple of hours before they are propagated globally.

9.5.1 AWS Route53

Route53 is a cloud-based DNS service by *Amazon Web Services (AWS)*. The technology is very similar to the *PowerDNS* example you just saw: *Route53* identifies the client IP address for incoming DNS requests and matches the requested hostname to an IP address that is associated with a specific geographic region.

Route53 can match continents, countries, and US states.

The following screenshot shows how to configure a DNS record with *geolocation routing*:



▼ Record 1

Record name Info: www .diy-cdn.varnish-software.com

Valid characters: a-z, 0-9, ! " # $ % & ' ( ) * + , - / : ; < = > ? @ [\ ] ^ _ ` { | } . ~

Record type Info: A – Routes traffic to an IPv4 address and ...

Value Info: 192.0.2.235 — Alias

Enter multiple values on separate lines.

TTL (seconds) Info: 300 — 1m 1h 1d

Recommended values: 60 to 172800 (two days)

Routing policy Info: Geolocation

Location: Europe

Health check - *optional* Info: Choose health check

Record ID Info: US West load balancer

AWS Route 53

The IP address that is returned represents the closest *CDN PoP* the user should connect to. If the *PoP nodes* are also hosted in the *AWS cloud*, *Route53* has some additional request routing capabilities.

9.5.1 Anycast

Anycast is a network-routing technique that maps a single IP address to multiple endpoints and lets routers decide which endpoint is selected.

Endpoint selection is based on the number of hops between the client and the endpoints, on distance, and network latency. *Anycast* will choose the shortest route.

Anycast may even select a *PoP* that is a lot further away because the latency is lower. *Geolocation* does not have this intelligence.

The preferred route for *Anycast addressing* is implemented using the *Border Gateway Protocol (BGP)*. This is a routing protocol that announces the routes over the network. This is not a *layer-7* implementation; however, it can be leveraged by *layer-7 protocols*, such as *HTTP* and *DNS*.

When routing traffic to a *CDN PoP*, *Anycast* can give you the IP address of a *load balancer* or an *edge-tier node*, which can be directly used by the *HTTP* protocol.

It is also possible that you use *Anycast* to send requests to specific *DNS servers*. The DNS server can then use finer-grained *geolocation* information that differs based on the selected DNS server.

9.5.1 Varnish Traffic Router

Varnish Software is also building a traffic router. The systems are designed to be perfectly compatible with its *Varnish Enterprise* offering and approaches the routing aspect from two different angles:

- DNS
- HTTP redirects

One big differentiator from the other request routing solutions is that the *Varnish Traffic Router* keeps track of *PoP* and *endpoint* utilization and health. It keeps track of *bandwidth consumption* and *request rate*. It takes the load of the individual *endpoints* and *PoP*s into account when routing traffic. An *endpoint* or *PoP* that is not healthy or overloaded will not get any traffic sent to it. There is also support for *CIDR* routing.

To avoid reinventing the wheel, *Varnish Traffic Router* doesn't implement a custom DNS server but leverages *PowerDNS* instead.

The *Varnish Traffic Router* uses *PowerDNS* to handle all *DNS* protocol specifics and acts as a *remote backend*. This means that *PowerDNS* polls an *HTTP endpoint* to retrieve zone information. This endpoint happens to be a specific listening port on the *Varnish Traffic Router*.

The logic, the rules, and the geolocation is done inside the traffic router.

This logic can also be exposed for incoming HTTP requests: when a client requests an HTTP resource on the traffic router, it decides based on the client IP address which *node* in a specific *PoP* is going to be selected. The result is an *HTTP 301* redirect to that node.

For websites, HTTP redirection is not ideal for *SEO* reasons. But for assets like images, video streams, and other static files, this is a viable solution. Video is the primary use case here.

Varnish Traffic Router is still an unreleased product, still in development at the time of writing this book. However, chances are that by the time you are reading this, the *Varnish Traffic Router* will be released, and its management integrated into the *Varnish Controller*.

9.6 Varnish and 5G

5G is coming; there's no doubt about it. There's plenty of hype surrounding this new cellular network technology.

The two metrics that are used to describe the benefits of *5G* are:

- Higher throughput
- Lower latency

In theory *5G* can be 100 times faster than *4G* with throughput up to *10 Gbit per second*. On average, *30 ms* of latency occurs on a *4G* connection. On *5G* this should be sub-millisecond.

And those happen to be terms you can associate with *Varnish* too. So how does *Varnish* fit into this story?

Higher throughput and lower latency will create expectations that are hard to meet: although your *5G* service may support throughput in the hundreds of megabits with a sub-millisecond latency, this doesn't mean that the content you're requesting will be received at the same pace.

The network route between the *5G antenna* and the server that contains the requested resources might not yield that kind of throughput and latency. But that's why *CDN*s were invented in the first place.

A centralized *CDN* with a number of *PoP*s will yield better results, but it will be nowhere near the theoretical numbers that are thrown around.

9.6.1 Multi-access edge computing

In a *5G* context, the only solution for better performance is to move *the edge* even closer to the user. This concept is part of *5G* and is called *multi-access edge computing (MEC)*.

Instead of relying on a centralized *cloud* for computing and storage, *5G* operators will run workloads in *5G edge locations*. These are *decentralized cloud environments* that are as close as possible to the *5G antennas*.

Only by putting the content even closer to the user than in a traditional *CDN architecture* can we truly improve the quality of experience for the user.

The *cloudification* of mobile technology, powered by open *radio access network (RAN)* standards, will lower the barriers to entry for developers and content providers and will lead to more *5G* integrations.

The *MEC* is a way for *ISP*s to build out a *CDN* deep in their network. This offloads their core network and provides ultra-low latency for clients in that area.

9.6.1 Use cases

The obvious use case that comes to mind is video: video is prone to latency, and delivering high-quality video requires plenty of bandwidth.

5G will push *OTT video streaming* to the next level:

- Higher video resolutions
- Higher frame rates
- Smaller video segments, reducing latency for live streaming
- 360° video

5G could also revolutionize gaming and accelerate the shift to gaming in the cloud. *Virtual reality (VR)* and *augmented reality (AR)* applications could also be pushed to the cloud thanks to *multi-access edge computing*.

The use cases are not limited to public mobile networks. Companies can build private *5G* networks, thanks to the open *RAN* standards and use those networks for industrial automation in their factories and plants.

Healthcare innovations powered by ultra-low latency robotic surgery could greatly benefit from *5G*.

And as *5G* promises to be 100 times faster than traditional mobile networks, this could also mean that traditional broadband subscriptions could be replaced with mobile subscriptions without jeopardizing the quality of experience.

9.6.1 Varnish Edge Cloud

With all those innovations on the horizon, *Varnish* is in an excellent position to add value to companies who want to build out a *MEC*.

Varnish Edge Cloud is the name of the our *5G* solution, which is of course based on *Varnish Enterprise*. As *5G* continues to evolve, so will *Varnish Edge Cloud*, to meet the evolving requirements for those building out *MEC*s. The virtualized nature of these setups will only further democratize mobile networking.

Here are some key capabilities that *Varnish Edge Cloud*, the *5G-branded version of Varnish Enterprise*, currently has:

- Being able to serve up to *800,000 requests per second*
- Supporting a throughput up to *200 Gbps*
- Latency below *one millisecond*
- The *Varnish Configuration Language* that allows developers to run workloads on the edge
- Clustering and high availability to synchronize caches across *MEC* locations
- Multi-terabyte *edge storage*, thanks to *MSE*

9.7 Summary

Content delivery has become an important topic for anyone that provides services on the web.

Increased latency and low throughput severely impact the quality of experience, which may drive them to your competitors.

Because of globalization, it is likely that people from all around the world are consuming your online services. Geography plays a significant part in the experience: the further your users are removed from the *origin*, the higher the risk of latency.

A key takeaway from this chapter is that *CDNs* are responsible for the fact that online content is delivered so fast and with such stability.

Without *CDNs*, *4K* video streams on your smart TV would be a lot more challenging.

Another takeaway is that it often makes sense to build your own *CDN*.

Varnish has some unique features that make it a suitable building block for *Private CDNs*.

Moving from building blocks to a ready-to-use product, *Varnish Enterprise* is equipped to deliver *enterprise-grade Private CDN* capabilities that are used by some of the biggest video streaming platforms, broadcasters and CDN providers in the world.

With the advent of *5G* and the need to push *the edge* even closer to the user, *Varnish* is again in a unique position to offer caching and edge computing solutions via *Varnish Software's Varnish Edge Cloud* solution.

Despite *Varnish Software*'s commitment to leverage its technology to build *Private CDN* solutions, it also possible to use *Varnish Cache* to build your own *CDN*.

It all depends on the storage needs you have, and which *VMOD*s your use case requires.

Not only does this wrap up *chapter 9*, we've also come full circle in the book. We'd like to invite you to turn the page, and read the closing notes of this book.

Chapter 10: Closing notes

Thank you for taking the time to read this book. Writing it was a memorable experience for me, and I hope it brought some inspiration and insight to you immediately and can be a solid reference resource for you in the future as well.

No doubt, we covered a lot of topics, and the scope of this book was pretty broad. This makes sense because this breadth illustrates just how varied the *Varnish ecosystem* is and paints a brighter, more detailed picture of Varnish being about much more than just writing a couple of lines of *VCL*.

Some of the key takeaways I hope you got from the book include:

Get started with and develop your hands-on knowledge of Varnish I sincerely hope that by now you are capable of setting up a cluster of *Varnish* servers, manage them properly, and perform the necessary monitoring and logging to ensure a stable *content delivery platform*.

Understand in greater depth how important HTTP is *Varnish* speaks HTTP, so it is important to know how the protocol works. Throughout the book, and especially in *chapter 3*, you've seen the ins and outs of *HTTP*, and you've learned all about its caching capabilities, and some of the specific headers that can be leveraged.

Appreciate that Varnish thrives on driving performance By now it should also be crystal clear that *Varnish* is built for performance. Some of the numbers I shared are quite impressive and make *Varnish* one of the fastest content delivery systems in the industry.

See the flexibility and power of the Varnish Configuration Language The unique selling point of *Varnish* is the *Varnish Configuration Language*: it is instrumental in offering the right caching experience, but it is even more important from an *edge-computing* point of view.

Yes, the *Varnish Configuration Language (VC)L* allows you to make very detailed decisions on when and how to cache content. But the true power of *VCL* lies in the fact that it can offload logic from the origin to the edge and cache otherwise uncacheable content.

Remember, *Varnish* is not necessarily a standalone piece of technology. The *Varnish ecosystem* is a lot broader than the `varnishd` program.

As you have learned in this book, there are also a collection of commercial solutions that are under the *Varnish Enterprise* umbrella, developed and maintained by *Varnish Software*.

First of all, there is *Varnish Enterprise* itself. This commercial caching solution adds native features, extra tools and a rich collection of enterprise *VMODs* on top of *Varnish Cache*.

In the end, *Varnish* can be considered *CDN technology* that powers the delivery of websites, APIs, OTT video streaming platforms, Private CDNs, and 5G multi-access edge clouds.

Web performance matters; content delivery matters. I hope we at *Varnish Software* have convinced you that *Varnish* is a great tool to address the modern content delivery challenges you may be facing.

10.1.1 Thank you

Dear Reader, I've already thanked you. But I also want to take the opportunity to thank my colleagues who have assisted me during the writing process.

They spent countless hours doing calls, assisting me in describing the ins and outs of *Varnish*, and coming up with interesting examples and valuable use cases.

As an author I'll probably get most of the credit. However, I would like to name everyone who was involved in the planning, writing and reviewing process:

- Alf-André Walla
- Alve Elde
- Andrew Wiik
- Arianna Aondio
- Asad Sajjad Ahmed
- Dag Haavi Finstad
- David Baron
- Dridi Boukelmoune
- Erik Tedfelt Lennström
- Erika Wolfe
- Espen Braastad
- Fredrik Steen
- Guillaume Quintard
- Henry Choi
- Ian Vaughan
- Kyle Simukka
- Lars Larsson
- Lucas Guardalben
- Magnus Persson
- Martin Blix Grydeland
- Miles Weaver
- Morten Bekkelund
- Niklas Brand
- Pål Hermunn Johansen
- Rein-Amund Schultz
- Reza Naghibi
- Sam Woodcock
- Steven Wojcik
- Torgeir Håpnes

Without these people, this book would be a lot less detailed and a lot less interesting. I can even say that there would be no book without these people.

10.1.2 What does the future bring?

The adoption of *Varnish* keeps increasing, and it is our aim to provide more and better resources for developers online as well as do more outreach to the open source community. Our ambition is that all organizations with content delivery and edge-compute needs should find Varnish technology to be the best solution.

Varnish Software is continuously growing as a company and attracting more customers that are looking for enterprise-grade content delivery and edge-compute solutions. This means that you can count on Varnish to continue developing even faster, richer functionality and offer availability on the most popular platforms.

The future of *Varnish Cache* and *Varnish Enterprise* is bright, and Varnish's importance in the web acceleration, content delivery, and 5G edge-compute space will continue to increase.

10.1.3 More information

If you would like to know more about *Varnish Cache*, please visit the official website at http://varnish-cache.org/.

For all things *Varnish Software*, you can visit https://www.varnish-software.com/.

If you're interested in more technical resources, *VCL examples* and tutorials, please visit our developer portal, which will be launched in 2021, at https://developers.varnish-software.com.

Varnish 6: By Example

Varnish is the leading web acceleration and content delivery software that powers more than ten million websites worldwide and that is trusted by more than 20% of the top 10,000 websites.

Varnish caches HTTP responses and accelerates anything that communicates over HTTP by doing so. The technology is used not only to accelerate websites and APIs, but also for live & VoD video streaming platforms, to build private CDNs, and even to operate in a 5G multi-access edge cloud context.

This book covers the ins and outs of the Varnish ecosystem and focuses on both the open source and enterprise versions of the software. This in-depth technical book is targeted at developers and operations engineers but can also be valuable for decision-makers.

Regardless of your role, if you're interested in high-performance content delivery, this book is for you.

Thijs Feryn is a technical evangelist at Varnish Software.
For more information, please visit **www.varnish-software.com.**