O'REILLY

Compliments of VARNISH SOFTWARE

Getting Started with Varnish Cache

ACCELERATE YOUR WEB APPLICATIONS

Thijs Feryn


ENGINEERED FOR
PERFORMANCE
Varnish Cache and Varnish Plus were engineered for speed and scalability. More than 2.5 million websites and 13% of the world's top websites trust Varnish to speed up their content delivery.
Learn why developers of some of the world's most recognized sites, from *The New York Times*, *Vimeo* and *Thomson Reuters* to *Twitch* and *CBC*, chose Varnish.
Start your free trial today: *https://info.varnish-software.com/varnish-plus-trial*
VARNISH SOFTWARE
varnish-software.com

Getting Started with Varnish Cache

Accelerate Your Web Applications

Thijs Feryn

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Getting Started with Varnish Cache

by Thijs Feryn

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Anderson and Virginia Wilson
Production Editor: Melanie Yarbrough
Copyeditor: Gillian McGarvey
Proofreader: Eliahu Sussman
Indexer: WordCo Indexing Services
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

February 2017: First Edition

Revision History for the First Edition

2017-02-01: First Release

978-1-491-97718-7

[LSI]

This book is dedicated to all the people who support me day in and day out:

My lovely wife Lize, my son Lex, and my daughter Lia. My mom, dad, sister, mother-in-law, and brothers-in-law.

And of course my friends—you know who you are.

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

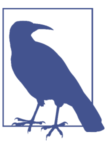
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.



This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

A big thank you to my employer Combell for granting me the time to write this book. More specifically, our CEO Jonas Dhaenens, my manager Frederik Poelman, and my colleagues Stijn Claerhout, Christophe Van den Bulcke, and Wesley Hof. Thanks for believing in me!

I would like to give a shout-out to Varnish Software for the opportunity to write my very first book. Thank you, Hildur Smaradottir, Per Buer, and Rubén Romero.

CHAPTER 1
What Is Varnish Cache?

Varnish Cache is a so-called *reverse caching proxy*. It's a piece of software that you put in front of your web server(s) to reduce the loading times of your website/application/API by caching the server's output. We're basically talking about web performance.

In this chapter, I'll explain why web performance is so important and how Varnish can improve it.

Why Does Web Performance Matter?

Many people underestimate the importance of web performance. The common logic is that if a website performs well when 10 users are accessing it, the site will also be fine when 1 million users want to access it. It only takes one successful marketing campaign to debunk that myth.

Performance and scalability aren't one and the same. Performance is the raw speed of your website: how many (milli)seconds does it take to load the page? Scalability, on the other hand, is keeping the performance stable when the load increases. The latter is a reason for bigger organizations to choose Varnish. The former applies to everyone, even small projects.

Let's say your website has about 100 visitors per day. Not that many, right? And the loading time of a page is 1.5 seconds—not great, but not that bad either. Without caching, it might take some time (and money) to reduce that loading time to less than a second. You might refactor your code or optimize your infrastructure. And then you might ask yourself if all of the effort was worth it.

It's also important to know that web performance is an essential part of the user experience. Want to please your users and ensure they stay on your site? Then make sure your pages are loading fast. Even Google knows this—did you know that Google Search takes the loading times of your website into account when calculating its page rank?

Poor performance will not only hurt your Google ranking, it will also impact your bottom line: people don't have the patience to wait for slow content and will look for an alternative in a heartbeat. In a heavily saturated market, they'll probably end up with one of your competitors.

Where Does Varnish Fit In?

With a correctly configured Varnish, you will automatically reduce the loading times of your website without much effort. Given that Varnish is open source and easy to set up, this is a no-brainer.

And if you play your cards right, who knows, maybe your site will become popular one day. The term "viral" comes to mind. If you already have a properly configured Varnish in place, you won't need to take many more measures.

A lot of people think that Varnish is technology for big projects and large companies—the kind of sites that attract massive amounts of hits. That's true; these companies do use Varnish. In fact, 13% of the top 10,000 websites rely on Varnish to ensure fast loading times. However, Varnish is also suitable for small and medium-sized projects. Have a look at Chapter 9 to learn about some of the success stories and business use cases.

All that being said, Varnish is not a silver bullet; it is only a part of the stack. Many more components are required to serve pages fast and reliably, even at load. These components, such as the network, server, operating system, web server, and the application runtime, can also fail on you.

The Varnish Cache Open Source Project

Varnish Cache is an open source project written in C. The fact that it's open source means the code is also available online and the use of Varnish is free of charge.

Varnish Cache is maintained by an active community, led by Poul-Henning Kamp. Although Varnish Cache is "free as in beer," there's still a company backing the project and funding most of its development. This company, called Varnish Software, is able to fund the Varnish Cache project by providing training, support, and extra features on top of Varnish.

At the time of writing, the most common version, which we will be covering in this book, is 4.1. Version 5 was released on September 15, 2016. However, this does not mean that this book is outdated. The adoption process for new versions takes a while.

How Does Varnish Work?

Varnish is either installed on web servers or on separate machines. Once installed and started, Varnish will mimic the behavior of the web server that sits behind it. Usually, Varnish listens on TCP port 80, the conventional TCP port that delivers HTTP—unless, of course, Varnish itself sits behind another proxy. Varnish will have one or more backends registered and will communicate with one of these backends in case a result cannot be retrieved from cache.

Varnish will preallocate a chunk of virtual memory and use that to store its objects. The objects contain the HTTP response headers and the payload that it receives from the backend. The objects stored in memory will be served to clients requesting the corresponding HTTP resource. The objects in cache are identified by a hash that, by default, is composed of the hostname (or the IP address if no hostname was specified) and the URL of the request.

Varnish is tremendously fast and relies on pthreads to handle a massive amount of incoming requests. The threading model and the use of memory for storage will result in a significant performance boost of your application. If configured correctly, Varnish Cache can easily make your website 1,000 times faster.

Varnish uses the *Varnish Configuration Language (VCL)* to control the behavior of the cache. VCL is a *domain-specific language* that offers hooks to override and extend the behavior of the different states in the Varnish Finite State Machine. These hooks are represented by a set of subroutines that exist in VCL. The subroutines and the VCL code live inside the VCL file. At startup time, the VCL file is read, translated to C, compiled, and dynamically loaded as a shared object.

The VCL syntax is quite extensive, but limited at some point. If you want to extend the behavior even further, you can write custom Varnish modules in C. These modules can contain literally anything you can program in C. This extended behavior is presented through a set of functions. These functions are exposed to VCL and enrich the VCL syntax.

VCL: The Heart and Soul of Varnish

VCL is the heart and soul of Varnish. It is the selling factor and the reason people prefer Varnish over other caching technologies. In Chapter 4, we'll cover VCL in great detail and in Chapter 9 you'll find some business use cases and success stories where Varnish and VCL saved the day.

One compelling example that comes to mind is a DDoS attack that was targeting WikiLeaks. The attack contained a clear pattern: the Accept headers were the same across all requests. A couple of lines of VCL were enough to fend off the attack.

It goes to show that the flexibility that VCL brings to the table is unparalleled in the world of caching.

Caching Is Not a Trick

The reality of the matter is that most websites, applications, and APIs are data-driven. This means that their main purpose is to present and visualize data that comes from the database or an external resource (feed, API, etc.). The majority of the time is spent on retrieving, assembling, and visualizing data.

When you don't cache, that process is repeated upon every client request. Imagine how many resources are wasted by recomputing, even though the data hasn't changed.

Back in the Day

Back in the day when I was still a student, my database teacher taught us all about database normalization and why we should always normalize data to the third normal form. He told us to never store results in the database that otherwise could be retrieved and recomputed. And he was right, at that time.

In those days, the load on a database server that was used to feed a website wasn't that high. However, hardware was more expensive. Storing computed results was not really a thing.

But as the web evolved, I had to question that statement. These days, my mantra is "Don't recompute if the data hasn't changed." And that, of course, is easier said than done.

If you decide to cache a computed result, you better have good control over the original data. If the original data does change, you will need to make sure the cache is updated. However, emptying the cache too frequently defies the purpose of the cache. It's safe to say that caching is a balancing act between serving up-to-date data and ensuring acceptable loading times.

Caching is not a trick, and it's not a way to compensate for poor performing systems or applications; caching is an architectural decision that, if done right, will increase efficiency and reduce infrastructure cost.

Conclusion

Slow websites suck. Users don't have much patience and in a highly-saturated market, having a fast website can give you the edge over your competitors. Raw performance is important, but a stable *time to first byte* under heavy load is just as important. We call this *scalability* and it's a tough nut to crack. There are plenty of ways to make your website scale, most of which require a considerable amount of time and money. Luckily, a decent caching strategy can reduce the impact of all that traffic. Varnish is a tool that can cache your HTTP traffic and take most of the load off your servers.

CHAPTER 2

Go, Go, Go and Get Started!

Now that you know what Varnish is all about, you're probably eager to learn how to install, configure, and use it. This chapter will cover the basic installation procedure on the most commonly supported operating systems and the typical configuration parameters that you can tune to your liking.

Installing Varnish

Varnish is supported on the following operating systems:

- Linux
- FreeBSD
- Solaris

You can get it to work on other UNIX-like systems (OS X, OpenBSD, NetBSD, and Windows with Cygwin), but there's no official support for those.

> In reality, you'll probably install Varnish on a Linux system. For development purposes, you might even run it on OS X. Linux is the most commonly used operating system for production systems. Some people do local development on a Mac and want to test their code locally. Therefore, it could make sense to install Varnish on OS X, just to see how your code behaves when it gets cached by Varnish.

The supported Linux distributions are:

- Ubuntu

- Debian
- Red Hat
- CentOS

You can easily install Varnish using the package manager of your operating system, but you can also compile Varnish from source.

Installing Varnish Using a Package Manager

Compiling from source is all fun and games, but it takes a lot of time. If you get one of the dependencies wrong or you install the wrong version of a dependency, you're going to have a bad day. Why bother doing it the hard way (unless you have your reasons) if you can easily install Varnish using the package manager of your operating system?

Here's a list of package managers you can use according to your operating system:

- APT on Ubuntu and Debian
- YUM on Red Hat and CentOS
- PKG on FreeBSD

> Even though FreeBSD officially supports Varnish, I will skip it for the rest of this book. In reality, few people run Varnish on FreeBSD. That doesn't mean I don't respect the project and the operating system, but I'm writing this book for the mainstream and let's face it: FreeBSD is not so mainstream.

Installing Varnish on Ubuntu and Debian

In simple terms, we can say that the Ubuntu and the Debian distributions are related. Ubuntu is a Debian-based operating system. Both distributions use the APT package manager. But even though the installation of Varnish is similar on both distributions, there are subtle differences. That's why there are different APT repository channels for Ubuntu and Debian.

Here's how you install Varnish on Ubuntu, assuming you're running the Ubuntu 14.04 LTS (Trusty Tahr) version:

```
apt-get install apt-transport-https
curl https://repo.varnish-cache.org/GPG-key.txt | apt-key add -
echo "deb https://repo.varnish-cache.org/ubuntu/ trusty varnish-4.1" \
     >> /etc/apt/sources.list.d/varnish-cache.list
apt-get update
apt-get install varnish
```

Packages are also available for other Ubuntu versions. Varnish only supports LTS versions of Ubuntu. Besides Trusty Tahr, you can also install Varnish on Ubuntu 12.04 LTS (Precise Pangolin) and Ubuntu 10.04 LTS (Lucid Lynx). You can do this by replacing the **`trusty`** keyword with either **`precise`** or **`lucid`** in the previous example.

If you're running Debian, here's how you can install Varnish on Debian 8 (Jessie):

```
apt-get install apt-transport-https
curl https://repo.varnish-cache.org/GPG-key.txt | apt-key add -
echo "deb https://repo.varnish-cache.org/debian/ jessie varnish-4.1"\
    >> /etc/apt/sources.list.d/varnish-cache.list
apt-get update
apt-get install varnish
```

If you're running an older version of Debian, there are packages available for Debian 5 (Lenny), Debian 6 (Squeeze), and Debian 7 (Wheezy). Just replace the **`jessie`** keyword with either **`lenny`**, **`squeeze`**, or **`wheezy`** in the preceding statements.

Installing Varnish on Red Hat and CentOS

There are three main distributions in the Red Hat family of operating systems:

- Red Hat Enterprise: the paid enterprise version
- CentOS: the free version
- Fedora: the bleeding-edge desktop version

All three of them have the YUM package manager, but we'll primarily focus on both Red Hat and CentOS, which have the same installation procedure.

If you're on Red Hat or CentOS version 7, here's how you install Varnish:

```
yum install epel-release
rpm --nosignature -i https://repo.varnish-cache.org/redhat/varnish-4.1.el7.rpm
yum install varnish
```

If you're on Red Hat or CentOS version 6, here's how you install Varnish:

```
yum install epel-release
rpm --nosignature -i https://repo.varnish-cache.org/redhat/varnish-4.1.el6.rpm
yum install varnish
```

Configuring Varnish

Now that you have Varnish installed on your system, it's time to configure some settings so that you can start using it.

Varnish has a bunch of startup options that allow you to configure the way you interact with it. These options are located in a configuration file and assigned to the `varnishd` program at startup time. Here are some examples of typical startup options:

- The address and port on which Varnish processes its incoming HTTP requests
- The address and port on which the Varnish CLI runs
- The location of the VCL file that holds the caching policies
- The location of the file that holds the secret key, used to authenticate with the Varnish CLI
- The storage backend type and the size of the storage backend
- Jailing options to secure Varnish
- The address and port of the backend that Varnish will interact with

> You can read more about the Varnish startup options on the official varnishd documentation page.

The Configuration File

The first challenge is to find where the configuration file is located on your system. This depends on the Linux distribution, but also on the service manager your operating system is running.

If your operating system uses the systemd service manager, the Varnish configuration file will be located in a different folder than it usually would be. Systemd is enabled by default on Debian Jessie and CentOS 7. Ubuntu Trusty Tahr still uses Sysv.

If you want to know where the configuration file is located on your operating system (given that you installed Varnish via a package manager), have a look at Table 2-1.

Table 2-1. Location of the Varnish configuration file

	SysV	Systemd
Ubuntu/Debian	/etc/default/varnish	/etc/systemd/system/varnish.service
Red Hat/CentOS	/etc/sysconfig/varnish	/etc/varnish/varnish.params

If you use systemd on Ubuntu or Debian, the */etc/systemd/system/varnish.service* configuration file will not yet exist. You need to copy it from */lib/systemd/system/*.

If you change the content of the configuration file, you need to reload the Varnish service to effectively load these settings. Run the following command to make this happen:

```
sudo service varnish reload
```

Some Remarks on Systemd on Ubuntu and Debian

If you're on Ubuntu or Debian and you're using the systemd service manager, there are several things you need to keep in mind.

First of all, you need to copy the configuration file to the right folder in order to override the default settings. Here's how you do that:

```
sudo cp /lib/systemd/system/varnish.service /etc/systemd/system
```

If you're planning to make changes to that file, don't forget that the results are cached in memory. You need to reload systemd in order to have your changes loaded from the file. Here's how you do that:

```
sudo systemctl daemon-reload
```

That doesn't mean Varnish will be started with the right startup options, only that systemd knows the most recent settings. You will still need to reload the Varnish service to load the configuration changes, like this:

```
sudo service varnish reload
```

Startup Options

By now you already know that the sole purpose of the configuration file is to feed the startup options to the varnishd program. In theory, you don't need a service manager: you can manually start Varnish by running varnishd yourself and manually assigning the startup options.

```
usage: varnishd [options]
    -a address[:port][,proto]    # HTTP listen address and port (default: *:80)
                                 #   address: defaults to loopback
                                 #   port: port or service (default: 80)
                                 #   proto: HTTP/1 (default), PROXY
    -b address[:port]            # backend address and port
                                 #   address: hostname or IP
                                 #   port: port or service (default: 80)
    -C                           # print VCL code compiled to C language
    -d                           # debug
```

```
-F                           # Run in foreground
-f file                      # VCL script
-h kind[,hashoptions]        # Hash specification
                             #   -h critbit [default]
                             #   -h simple_list
                             #   -h classic
                             #   -h classic,<buckets>
-i identity                  # Identity of varnish instance
-j jail[,jailoptions]        # Jail specification
                             #   -j unix[,user=<user>][,ccgroup=<group>]
                             #   -j none
-l vsl[,vsm]                 # Size of shared memory file
                             #   vsl: space for VSL records [80m]
                             #   vsm: space for stats counters [1m]
-M address:port              # Reverse CLI destination
-n dir                       # varnishd working directory
-P file                      # PID file
-p param=value               # set parameter
-r param[,param...]          # make parameter read-only
-S secret-file               # Secret file for CLI authentication
-s [name=]kind[,options]     # Backend storage specification
                             #   -s malloc[,<size>]
                             #   -s file,<dir_or_file>
                             #   -s file,<dir_or_file>,<size>
                             #   -s file,<dir_or_file>,<size>,<granularity>
                             #   -s persistent (experimental)
-T address:port              # Telnet listen address and port
-t TTL                       # Default TTL
-V                           # version
-W waiter                    # Waiter implementation
                             #   -W epoll
                             #   -W poll
```

The varnishd documentation page has more detailed information about all of the startup options.

Let's take a look at some of the typical startup options you'll encounter when setting up Varnish. The examples I use represent the ones coming from */etc/default/varnish* on an Ubuntu system that uses Sysv as the service manager.

Common startup options

The list of configurable startup options is quite extensive, but there's a set of common ones that are just right to get started. The following example does that:

```
DAEMON_OPTS="-a :80 \
             -a :81,PROXY \
             -T localhost:6082 \
             -f /etc/varnish/default.vcl \
             -S /etc/varnish/secret \
             -s malloc,3g \
             -j unix,user=www-data \"
```

Network binding. The most essential networking option is the `-a` option. It defines the address, the port, and the protocol that are used to connect with Varnish. By default, its value is `:6081`. This means that Varnish will be bound to all available network interfaces on TCP port 6081. In most cases, you'll immediately switch the value to `80`, the conventional HTTP port.

You can also decide which protocol to use. By default, this is HTTP, but you can also set it to PROXY. The PROXY protocol adds a so-called "preamble" to your TCP connection and contains the real IP address of the client. This only works if Varnish sits behind another proxy server that supports the PROXY protocol. The PROXY protocol will be further discussed in "What About TLS/SSL?" on page 16.

You can define multiple listening addresses by using multiple `-a` options. Multiple listening addresses can make sense if you're combining HTTP and PROXY support, as previously illustrated.

CLI address binding. The second option we will discuss is the `-T` option. It is used to define the address and port on which the Varnish CLI listens. In "Banning from the Command Line" on page 71, we'll need CLI access to invalidate the cache.

By default, the Varnish CLI is bound to `localhost` on port 6082. This means the CLI is only locally accessible.

> Be careful when making the CLI remotely accessible because although access to the CLI requires authentication, it still happens over an unencrypted connection.

Security options. The `-j` option allows you to jail your Varnish instance and run the subprocesses under the specified user. By default, all processes will run using the *varnish* user.

The jailing option is especially useful if you're running multiple Varnish instances on a single server. That way, there is a better process isolation between the instances.

The `-S` option is used to define the location of the file that contains the secret key. This secret key is used to authenticate with the Varnish CLI. By default, the location of this file is */etc/varnish/secret*. It automatically contains a random value.

You can choose not to include the `-S` parameter to allow unauthenticated access to the CLI, but that's something I would strongly advise against. If you want to change the location of the secret key value, change the value of the `-S` parameter. If you just want to change the secret key, edit */etc/varnish/secret* and reload Varnish.

Storage options. Objects in the cache need to be stored somewhere. That's where the `-s` option comes into play. By default, the objects are stored in memory *(~malloc)* and the size of the cache is 256 MiB.

> Varnish expresses the size of the cache in kibibytes, mebibytes, gibibytes, and tebibytes. These differ from the traditional kilobytes, megabytes, gigabytes, and terrabytes. The "bi" in kibibytes stands for binary, so that means a kibibyte is 1,024 bytes, whereas a kilobyte is 1,000 bytes. The same logic applies to mebibytes (1024 × 1,024 bytes), gibibytes (1024 × 1024 × 1024 bytes), and tebibytes (1024 × 1024 × 1024 × 1024 bytes).

The size of your cache and the storage type heavily depends on the number of objects you're going to store. If all of your cacheable files fit in memory, you'll be absolutely fine. Memory is fast and simple, but unfortunately, your memory will be limited in terms of size. If your Varnish instance runs out of memory, it will apply a so-called *Least Recently Used* (LRU) strategy to evict items from cache.

> If you don't specify the size of the storage and only mention malloc, the size of the cache will be unlimited. That means Varnish could potentially eat all of your server's memory. If your server runs out of memory, it will use the operating system's swap space. This basically stores the excess data on disk. This could cause a major slowdown of your entire system if your disks are slow.

Varnish counts the amount of hits per cached object. When it has to evict objects due to a lack of available memory, it will evict the least popular objects until it has enough space to store the next requested object.

If you have a dedicated Varnish server, it is advised to allocate about 80% of your available memory to Varnish. That means you'll have to change the `-s` startup option.

> File storage is also supported. Although it is slower than memory, it will still be buffered in memory. In most cases, memory storage will do the trick for you.

VCL file location. The location of the VCL file is set using the `-f` option. By default it points to */etc/varnish/default.vcl*. If you want to switch the location of your VCL file to another file, you can modify this option.

If you do not specify an `-f` option, you will need to add the `-b` option to define the backend server that Varnish will use.

Going more advanced

Let's turn it up a notch and throw some more advanced startup options into the mix. Here's an example:

```
DAEMON_OPTS="-a :80 \
             -a :81,PROXY \
             -T localhost:6082 \
             -f /etc/varnish/default.vcl \
             -S /etc/varnish/secret \
             -s malloc,3g \
             -j unix,user=www-data \
             -l 100m,10m \
             -t 60 \
             -p feature=_++esi_disable_xml_check \
             -p connect_timeout=5 \
             -p first_byte_timeout=10 \
             -p between_bytes_timeout=2"
```

Shared log memory storage. Varnish doesn't just store its objects; there's also space allocated in memory for logging and statistics. This information is used by utility binaries like `varnishlog`, `varnishtop`, and `varnishstat`.

By default, 1 MiB is allocated to the Varnish Statistics Counters (VSC) and 81 MiB is allocated to the Varnish Shared Memory Logs (VSL).

You can manipulate the size of the VSC and the VSL by changing the value of the `-l` startup option.

Default time-to-live. Varnish relies on expires or cache-control headers to determine the time-to-live of an object. If no headers are present and no explicit time-to-live was specified in the VCL file, Varnish will default to a time-to-live of 120 seconds. You can modify the default time-to-live at startup time by setting the `-t` startup option. The value of this option is expressed in seconds.

Runtime parameters. There are a bunch of runtime parameters that can be tuned. Overriding a runtime parameter is done by setting the `-p` startup option. Alternatively, if you want these parameters to be read-only, you can use the `-r` option. Setting parameters to read-only restricts users with Varnish CLI access from overriding them at runtime.

Have a look at the full list of runtime parameters on the varnishd documentation page.

In the preceding example, we're setting the following runtime parameters:

- `feature=esi_disable_xml_check`
- `connect_timeout`
- `first_byte_timeout`
- `between_bytes_timeout`

The first one (`feature=esi_disable_xml_check`) disables XML checks during the Edge Side Includes (ESI) processing. By default, Varnish requires ESI content to be valid XML. This is not always ideal, as this setting removes XML validation. ESI is a technique used by Varnish to assemble a page containing content blocks that come from multiple URLs. Each include can have its own time-to-live that is respected by Varnish. Varnish assembles content from the URLs using ESI include tags like `<esi:include src="http://example.com" />`. ESI allows you to still cache parts of a page that would otherwise be uncacheable (more information on ESI in "Edge Side Includes" on page 104).

The second one sets the `connect_timeout` to five seconds. This means that Varnish will wait up to five seconds when connecting with the backend. If the timeout is exceeded, a backend error is returned. The default value is 3.5 seconds.

The third one sets the `first_byte_timeout` to 10 seconds. After establishing a connection with the backend, Varnish will wait up to 10 seconds until the first byte comes in from the backend. If that doesn't happen within 10 seconds, a backend error is returned. The default value is 60 seconds.

The fourth one sets the `between_bytes_timeout` to two seconds. When data is returned from the backend, Varnish expects a constant byte flow. If Varnish has to wait longer than two seconds between bytes, a backend error is returned. The default value is 60 seconds.

What About TLS/SSL?

Transport Layer Security (TLS), also referred to as Secure Sockets Layer (SSL), is a set of cryptographic protocols that are used to encrypt data communication over the network. In a web context, TLS and SSL are the "S" in HTTPS. TLS ensures that the connection is secured by encrypting the communication and establishing a level of trust by issuing certificates.

During the last couple of years, TLS has become increasingly popular to the point that non-encrypted HTTP traffic will no longer be considered normal in a couple of years. Security is still a hot topic in the IT industry, and nearly every brand on the internet wants to show that they are secure and trustworthy by offering HTTPS on their sites. Even Google Search supposedly gives HTTPS websites a better page rank.

> The Varnish project itself hasn't included TLS support in its code base. Does that mean you cannot use Varnish in projects that require TLS? Of course not! If that were the case, Varnish's days would be numbered in the low digits.

Varnish does not natively include TLS support because encryption is hard and it is not part of the project's core business. Varnish is all about caching and leaves the crypto to the crypto experts.

The trick with TLS on Varnish is to terminate the secured connection before the traffic reaches Varnish. This means adding a TLS/SSL offloader to your setup that terminates the TLS connection and communicates over HTTP with Varnish.

The downside is that this also adds another layer of complexity to your setup and another system that can fail on you. Additionally, it's a bit harder for the web server to determine the origin IP address. Under normal circumstances, Varnish should add the value of the X-Forwarded-For HTTP request header sent by the TLS offloader and store that value in its own X-Forwarded-For header. That way, the backend can still retrieve the origin IP.

In Varnish 4.1, PROXY protocol support was added. The PROXY protocol is a small protocol that was introduced by HAProxy, the leading open source load-balancing software. This PROXY protocol adds a small preamble to the TCP connection that contains the IP address of the original client. This information is transferred along and can be interpreted by Varnish. Varnish will use this value and automatically add it to the X-Forwarded-For header that it sends to the backend.

I wrote a detailed blog post about this, and it contains more information about both the HAProxy and the Varnish setup.

Additionally, the PROXY protocol implementation in Varnish uses this new origin IP information to set a couple of variables in VCL:

- It sets the `client.ip` variable to the IP address that was sent via the PROXY protocol
- It sets the `server.ip` variable to the IP address of the server that accepted the initial connection
- It sets the `local.ip` variable to the IP address of the Varnish server
- It sets the `remote.ip` variable to the IP address of the machine that sits in front of Varnish

HAProxy is not the only TLS offloader that supports PROXY. Varnish Software released Hitch, a TLS proxy that terminates the TLS connection and communicates over HTTP with Varnish. Whereas HAProxy is primarily a load balancer that offers TLS offloading, Hitch only does TLS offloading. HAProxy also wrote a blog post about the subject that lists a set of PROXY-protocol ready projects. Depending on your use case and whether you need load balancing in your setup, you can choose either HAProxy or a dedicated TLS proxy. Varnish Plus, the advanced version of Varnish, developed by Varnish Software, offers TLS/SSL support on both the server and the client side. The TLS/SSL proxy in Varnish Plus is tightly integrated with Varnish and helps improve website security without relying on third-party solutions.

Conclusion

Don't let all these settings scare you—they're just proof that Varnish is an incredibly flexible tool with lots of options and settings that can be tuned.

If you're a sysadmin, I hope I have inspired you to try tuning some of these settings. If you're not, just remember that Varnish can easily be installed with a package manager of your Linux distribution and hardly requires any tuning to be up and running.

At the bare minimum, have a look at the setting in "Network binding" on page 13 if you want Varnish to process HTTP traffic on port 80.

CHAPTER 3
Varnish Speaks HTTP

Now that we have set up Varnish, it's time to use it. In Chapter 2 we talked about the configuration settings, so by now you should have the correct networking settings that allow you to receive HTTP requests either directly on port 80 or through another proxy or load balancer.

Out-of-the-box Varnish can already do a lot for you. There is a default behavior that is expressed by the built-in VCL and there are a set of rules that Varnish follows. If your backend application complies with these rules, you'll have a pretty decent hit rate.

> Varnish uses a lot of HTTP best practices to decide what gets cached, how it gets cached, and how long it gets cached. As a web developer, I strongly advise that you apply these best practices in the day-to-day development of your backend applications. This empowers you and helps you avoid having to rely on custom Varnish configurations that suit your application. It keeps the caching logic portable.

Unlike many other proxies, Varnish is an *HTTP accelerator*. That means Varnish does HTTP and HTTP only. So it makes sense to know HTTP and how it behaves.

There are five ways in which Varnish respects HTTP best practices:

- Idempotence
- State
- Expiration
- Conditional requests

- Cache variations

Let's have a look at each of these and explore how Varnish deals with them.

Idempotence

Varnish will only cache resources that are requested through an *idempotent HTTP verb*, which are HTTP verbs that do not change the state of the resource. To put it simply, Varnish will only cache requests using the following methods:

- GET
- HEAD

And that makes perfect sense: if you issue a request using `POST` or `PUT`, the method itself implies that a change will happen. In that respect, caching wouldn't make sense because you would be caching stale data right from the get-go.

So if Varnish sees a request coming in through, let's say, `POST`, it will pass the request to the backend and will not cache the returned response.

For the sake of completeness, these are the HTTP verbs/methods that Varnish can handle:

- `GET` (can be cached)
- `HEAD` (can be cached)
- `PUT` (cannot be cached)
- `POST` (cannot be cached)
- `TRACE` (cannot be cached)
- `OPTIONS` (cannot be cached)
- `DELETE` (cannot be cached)

All other HTTP methods are considered non-RFC2616 compliant and will completely bypass the cache.

Although I'm referring to the RFC2616, this RFC is, in fact, dead and was replaced by the following RFCs:

- RFC7230
- RFC7231
- RFC7232
- RFC7233
- RFC7234
- RFC7235

State

Now that you know about idempotence and how HTTP request methods shouldn't change the state of the resource, let's look at other mechanisms in HTTP that can control state. I'm not talking about global state, but more specifically about *user-specific* data. There are two ways to keep track of state for users:

- Authorization headers
- Cookies

Whenever Varnish sees one of these, it will pass the request off to the backend and not cache the response. This happens because when an authentication header or a cookie is sent, it implies that the data will differ for each user performing that request.

If you decide to cache the response of a request that contains an authentication header or cookie, you would be serving a response tailored to the first user that requested it. Other users will see it, too, and the response could potentially contain sensitive or irrelevant information.

But let's face it: cookies are our main instrument to keep track of state, and websites that do not uses cookies are hard to come by. Unfortunately, the internet uses too many cookies and often for the wrong reasons.

We use cookies to establish sessions in our application. We can also use cookies to keep track of language, region, and other preferences. And then there are the tracking cookies that are used by third parties to "spy" on us.

In terms of HTTP, cookies appear both in the request and the response process. It is the backend that sets one or more cookies by issuing a `Set-Cookie` response header. The client receives that response and stores the cookies in its local cookie store.

As you can see in the example below, a cookie is a set of key-value pairs, delimited by an ampersand.

```
Set-Cookie: language=en&country=us
```

When a client has stored cookies for a domain, it will use a `Cookie` request header to send the cookies back to the server upon every subsequent request. The cookies are also sent for requests that do not require a specific state (e.g., static files).

```
Cookie: language=en&country=us
```

This two-step process is how cookies are set and announced. Just remember the difference between `Cookie` and `Set-Cookie`. The first is a request header; the second is a response header.

> I urge web developers to not overuse cookies. Do not initiate a session that triggers a `Set-Cookie` just because you can. Only set sessions and cookies when you really need to. I know it's tempting, but consider the impact.

As mentioned, Varnish doesn't like to cache cookies. Whenever it sees a request with a `Cookie` header, the request will be passed to the backend and the response will not be cached.

When a request does not contain a cookie but the response includes a `Set-Cookie` header, Varnish will not store the result in cache.

Expiration

HTTP has a set of mechanisms in place to decide when a cached object should be removed from cache. Objects cannot live in cache forever: you might run out of cache storage (memory or disk space) and Varnish will have to evict items using an LRU strategy to clear space. Or you might run into a situation where the data you are serving is stale and the object needs to be synchronized with a new response from the backend.

Expiration is all about setting a time-to-live. HTTP has two different kinds of response headers that it uses to indicate that:

Expires
: An absolute timestamp that represents the expiration time.

Cache-control
: The amount of seconds an item can live in cache before becoming stale.

Varnish gives you a heads-up regarding the age of a cached object. The `Age` header is returned upon every response. The value of this `Age` header corresponds to the amount of time the object has been in cache. The actual time-to-live is the cache lifetime minus the age value. For that reason, I advise you not to set an `Age` header yourself, as it will mess with the TTL of your objects.

The Expires Header

The `Expires` header is a pretty straight forward one: you just set the date and time when an object should be considered stale. This is a response header that is sent by the backend.

Here's an example of such a header:

```
Expires: Sat, 09 Sep 2017 14:30:00 GMT
```

Do not overlook the fact that the time of an `Expires` header is based on Greenwich Mean Time. If you are located in another time zone, please express the time accordingly.

The Cache-Control Header

The `Cache-control` header defines the time-to-live in a relative way: instead of stating the time of expiration, `Cache-control` states the amount of seconds until the object expires. In a lot of cases, this is a more intuitive approach: you can say that an object should only be cached for an hour by assigning 3,600 seconds as the time-to-live.

This HTTP header has more features than the `Expires` header: you can set the time to live for both clients and proxies. This allows you to define distinct behavior depending on the kind of system that processes the header; you can also decide whether to cache and whether to revalidate with the backend.

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

The preceding example uses three important keywords to define the time-to-live and the ability to cache:

`public`
: Indicates that *both browsers and shared caches* are allowed to cache the content.

`max-age`
: The time-to-live in seconds that must be respected by the *browser*.

`s-maxage`
: The time-to-live in seconds that must be respected by the *proxy*.

It's also important to know that Varnish only respects a subset of the `Cache-control` syntax. It will only respect the keywords that are relevant to its role as a reverse caching proxy:

- `Cache-control` headers sent by the browser are ignored
- The time-to-live from an `s-maxage` statement is prioritized over a `max-age` statement
- `Must-revalidate` and `proxy-revalidate` statements are ignored
- When a `Cache-control` response header contains the terms private, no-cache, or no-store, the response is not cached

> Although Varnish respects the `public` and `private` keywords, it doesn't consider itself a shared cache and exempts itself from some of these rules. Varnish is more like a *surrogate web server* because it is under full control of the web server and does the webmaster's bidding.

Expiration Precedence

Varnish respects both `Expires` and `Cache-control` headers. In the Varnish Configuration Language, you can also decide what the time-to-live should be regardless of caching headers. And if there's no time-to-live at all, Varnish will fall back to its hardcoded default of 120 seconds.

Here's the list of priorities that Varnish applies when choosing a time-to-live:

1. If `beresp.ttl` is set in the VCL, use that value as the time-to-live.
2. Look for an `s-maxage` statement in the `Cache-control` header.
3. Look for a `max-age` statement in the `Cache-control` header.
4. Look for an `expires` header.
5. Cache for 120 seconds under all other circumstances.

> As you can see, the TTL in the VCL gets the absolute priority. Keep that in mind, because this will cause any other `Expires` or `Cache-control` header to be ignored in favor of the `beresp.ttl` value.

Conditional Requests

Expiration is a valuable mechanism for updating the cache. It's based on the concept of checking the *freshness* of an object at set intervals. These intervals are defined by the time-to-live and are processed by Varnish. The end user doesn't really have a say in this.

After the expiration, both the headers and the payload are transmitted and stored in cache. This could be a very resource-intensive matter and a waste of bandwidth, especially if the requested data has not changed in that period of time.

Luckily, HTTP offers a way to solve this issue. Besides relying on a time-to-live, HTTP allows you to keep track of the validity of a resource. There are two separate mechanisms for that:

- The `Etag` response header
- The `Last-Modified` response header

> Most web browsers support conditional requests based on the `Etags` and `Last-Modified` headers, but Varnish supports this as well when it communicates with the backend.

ETag

An `Etag` is an HTTP response header that is either set by the web server or your application. It contains a unique value that corresponds to the state of the resource.

A common strategy is to create a unique hash for that resource. That hash could be an `md5` or a `sha` hash based on the URL and the internal modification date of the resource. It could be anything as long as it's unique.

```
HTTP/1.1 200 OK
Host: localhost
Etag: 7c9d70604c6061da9bb9377d3f00eb27
Content-type: text/html; charset=UTF-8

Hello world output
```

As soon as a browser sees this `Etag`, it stores the value. Upon the next request, the value of the `Etag` will be sent back to the server in an `If-None-Match` request header.

```
GET /if_none_match.php HTTP/1.1
Host: localhost
User-Agent: curl/7.48.0
If-None-Match: 7c9d70604c6061da9bb9377d3f00eb27
```

The server receives this `If-None-Match` header and checks if the value differs from the `Etag` it's about to send.

If the `Etag` value is equal to the `If-None-Match` value, the web server or your application can return an `HTTP/1.1 304 Not Modified` response header to indicate that the value hasn't changed.

```
HTTP/1.0 304 Not Modified
Host: localhost
Etag: 7c9d70604c6061da9bb9377d3f00eb27
```

When you send a 304 status code, you don't send any payload, which can dramatically reduce the amount of bytes sent over the wire. The browser receives the 304 and knows that it can still output the old data.

If the `If-None-Match` value doesn't match the `Etag`, the web server or your application will return the full payload, accompanied by the `HTTP/1.1 200 OK` response header and, of course, the new `Etag`.

This is an excellent way to conserve resources. Whereas the primary goal is to reduce bandwidth, it will also help you to reduce the consumption of memory, CPU cycles, and disk I/O if you implement it the right way.

Here's an implementation example. It's just some dummy script that, besides proving my point, serves no real purpose. It's written in PHP because PHP is my language of choice. The implementation is definitely not restricted to PHP. You can implement this in any server-side language you like.

```
<?php
$etag = md5(__FILE__.filemtime(__FILE__));
header('Etag: ' . $etag);
if (isset($_SERVER['HTTP_IF_NONE_MATCH'])
&&  $_SERVER['HTTP_IF_NONE_MATCH'] == $etag) {
    header('HTTP/1.0 304 Not Modified');
    exit;
}
sleep(5);
?>
<h1>Etag example</h1>
<?php
echo date("Y-m-d H:i:s").'<br />';
```

Last-Modified

ETags aren't the only way to do conditional requests; there's also an alternative technique based on the `Last-Modified` response header. The client will then use the `If-Modified-Since` request header to validate the freshness of the resource.

The approach is similar:

1. Let your web server or application return a `Last-Modified` response header
2. The client stores this value and uses it as an `If-Modified-Since` request header upon the next request
3. The web server or application matches this `If-Modified-Since` value to the modification date of the resource
4. Either an `HTTP/1.1 304 Not Modified` or a `HTTP/1.1 200 OK` is returned

The benefits are the same: reduce the bytes over the wire and load on the server by avoiding the full rendering of output.

> The timestamps are based on the GMT time zone. Please make sure you convert your timestamps to this time zone to avoid weird behavior.

The starting point in the following example is the web server (or the application) returning a `Last-Modified` response header:

```
HTTP/1.1 200 OK
Host: localhost
Last-Modified: Fri, 22 Jul 2016 10:11:16 GMT
Content-type: text/html; charset=UTF-8

Hello world output
```

The browser stores the `Last-Modified` value and uses it as an `If-Last-Modified` in the next request:

```
GET /if_last_modified.php HTTP/1.1
Host: localhost
User-Agent: curl/7.48.0
If-Last-Modified: Fri, 22 Jul 2016 10:11:16 GMT
```

The resource wasn't modified, a 304 is returned, and the `Last-Modified` value remains the same:

```
HTTP/1.0 304 Not Modified
Host: localhost
Last-Modified: Fri, 22 Jul 2016 10:11:16 GMT
```

The browser does yet another conditional request:

```
GET /if_last_modified.php HTTP/1.1
Host: localhost
User-Agent: curl/7.48.0
If-Last-Modified: Fri, 22 Jul 2016 10:11:16 GMT
```

The resource was modified in the meantime and a full 200 is returned, including the payload and a new `Last-Modified`_header.

```
HTTP/1.1 200 OK
Host: localhost
Last-Modified: Fri, 22 Jul 2016 11:00:23 GMT
Content-type: text/html; charset=UTF-8

Some other hello world output
```

Time for another implementation example for conditional requests, this time based on the `Last-Modified` header. Again, it's dummy code, written in PHP:

```
<?php
header('Last-Modified: ' .
gmdate('D, d M Y H:i:s', filemtime(__FILE__)) . ' GMT');
if (isset($_SERVER['HTTP_IF_MODIFIED_SINCE']) &&
    strtotime($_SERVER['HTTP_IF_MODIFIED_SINCE']) >= filemtime(__FILE__))
{
    header('HTTP/1.0 304 Not Modified');
    exit;
}
sleep(5);
?>
<h1>Last-Modified example</h1>
<?php
echo date("Y-m-d H:i:s").'<br />';
```

Just like in the previous implementation example, we fake the delay caused by heavy load and use a `sleep` statement to make the application seem slower than it really is.

How Varnish Deals with Conditional Requests

When Varnish spots an `If-Modified-Since` or `If-None-Match` header in the request, it keeps track of the `Last-Modified` timestamp and/or the `Etag`. Regardless of whether or not Varnish has the object in cache, a 304 status code will be returned if the `Last-Modified` or the `Etag` header matches.

From a client point of view, Varnish reduces the amount of bytes over the wire by returning the 304.

On the other hand, Varnish also supports conditional requests when it comes to backend communication: when an object is considered stale, Varnish will send `If-Modified-Since` and `If-None-Match` headers to the backend if the previous response from the backend contained either a `Last-Modified` timestamp or an `Etag`.

When the backend returns a 304 status code, Varnish will not receive the body of that response and will assume the content hasn't changed. As a consequence, the stale data will have been revalidated and will no longer be stale. The `Age` response header will be reset to zero and the object will live in cache in accordance to the time-to-live that was set by the web server or the application.

Typically, stale data is revalidated by Varnish, but there is a VCL variable that allows you to manipulate that behavior: the `beresp.keep` variable decides how long stale objects will be returned while performing a conditional request. It's basically an amount of time that is added to the time-to-live. This allows Varnish to perform the conditional requests asynchronously without the client noticing any delays. The `beresp.keep` variable works independently from the `beresp.grace` variable.

> Both `beresp.keep` and `beresp.grace`, as well as many other VCL objects and variables, will be discussed in Chapter 4.

Cache Variations

In general, an HTTP resource is public and has the same value for every consumer of the resource. If data is user-specific, it will, in theory, not be cacheable. However, there are exceptions to this rule and HTTP has a mechanism for this.

HTTP uses the `Vary` header to perform cache variations. The `Vary` header is a response header that is sent by the backend. The value of this header contains the name of a request header that should be used to vary on.

> The value of the `Vary` header can only contain a valid request header that was set by the client. You can use the value of custom `X-` HTTP headers as a cache variation, but then you need to make sure that they are set by the client.

A very common example is language detection based on the `Accept-Language` request header. Your browser will send this header upon every request. It contains a set of languages or locales that your browser supports. Your application can then use the value of this header to determine the language of the output. If the desired language is not exposed in the URL or through a cookie, the only way to know is by using the `Accept-Language` header.

If no vary header is set, the cache (either the browser cache or any intermediary cache) has no way to identify the difference and stores the object based on the first request. If that first request was made in Dutch, all other users will get output in Dutch—regardless of the browser language—for the duration of the cache lifetime.

That is a genuine problem, so in this case, the application returns a `Vary` header containing `Accept-Language` as its value. Here's an example:

The browser language is set to Dutch:

```
GET / HTTP/1.1
Host: localhost
Accept-Language: nl
```

The application sets a `Vary` header that instructs the cache to keep a separate version of the cached object based on the `Accept-Language` value of the request.

```
HTTP/1.1 200 OK
Host: localhost
Vary: Accept-Language

Hallo, deze pagina is in het Nederlands geschreven.
```

The cache knows there is a Dutch version of this resource and will store it separately, but it will still link it to the cached object of the main resource. When the next request is sent from a browser that only supports English, the cached object containing Dutch output will not be served. A new backend request will be made and the output will be stored separately.

> Be careful when you peform cache variations based on request headers that can contain many different values. The `User-Agent` and the `Cookie` headers are perfect examples.
>
> In many cases, you don't have full control over the cookie value. Tracking cookies set by third-party services can add unique values per user to the cookie. This could result in too many variations, and the hit rate would plummet.
>
> The same applies to the `User-Agent`: almost every device has its own `User-Agent`. When using this as a cache variation, the hit rate could drop quite rapidly.

Varnish respects the `Vary` header and adds variations to the cache on top of the standard identifiers. The typical identifiers for a cached object are the hostname (or the IP if no hostname was set) and the URL.

When Varnish notices a cache variation, it will create a cache object for that version. Cache variations can expire separately, but when the main object is invalidated, the variations are gone, too.

You have to find a balance between offering enough cache variations and a good hit rate. Choose the right request header to vary on and look for balance.

Varnish Built-In VCL Behavior

Now that we know how Varnish deals with HTTP, we can summarize how Varnish behaves right out of the box. Here's a set of questions we can ask ourselves:

1. When is a request considered cacheable in Varnish?
2. When does Varnish completely bypass the cache?
3. How does Varnish identify an object?
4. When does Varnish cache an object?
5. What happens if an object is not stored in cache?
6. How long does Varnish cache an object?

Sounds mysterious, huh? Let me provide answers and allow me to explain how Varnish respects HTTP best practices.

When Is a Request Considered Cacheable?

When Varnish receives a request, it has to decide whether or not the response can be cached or even served from cache. The rules are simple and based on idempotence and state.

A request is *cacheable* when:

- The request method is *GET* or *HEAD*
- There are no cookies being sent by the client
- There is no authorization header being sent

When these criteria are met, Varnish will look the resource up in cache and will decide if a backend request is needed, or if the response can be served from cache.

When Does Varnish Completely Bypass the Cache?

If a request is not cacheable, the request is passed, a backend connection is made and the result is stored in the hit-for-pass cache. An example of this is a `POST` request.

But all of this happens under the assumption that the request method is a valid one that complies to RFC2616. Other request methods will not be processed by Varnish and will be piped to the backend.

When Varnish goes in to *pipe mode*, it opens a TCP connection to the backend, transmits the original request and immediately returns the response. There's no further processing of the request or response.

Here's a list of valid request methods according to the built-in VCL:

- GET
- HEAD
- PUT
- POST
- DELETE
- TRACE
- OPTIONS

All other request methods will be piped to the backend.

> RFC 2616 does not support request methods like `PATCH`, `LINK`, or `UNLINK`. Those were introduced in RFC 2068. If you require support for either of those methods, you'll need to customize your *VCL* and include those methods.
>
> "A Real-World VCL File" on page 62 offers a solution for that.

How Does Varnish Identify an Object?

Once we decide that an object is cacheable, we need a way to identify the object in order to retrieve it from cache. A hash key is composed of several values that serve as a unique identifier.

1. If the request contains a `Host` header, the hostname will be added to the hash.
2. Otherwise, the IP address will be added to the hash.
3. The URL of the request is added to the hash.

Based on that hash, Varnish will retrieve the object from cache.

When Does Varnish Cache an Object?

If an object is not stored in cache or when it's considered stale, a backend connection is made. Based on the backend response, Varnish will decide if the returned object will be stored in cache or if the cache is going to be bypassed.

A response will be stored in cache when:

- The time-to-live is more than zero.
- The response doesn't contain a `Set-Cookie` header.
- The `Cache-control` header doesn't contain the terms no-cache, no-store, or private.
- The `Vary` header doesn't contain `*`, meaning vary on all headers.

What Happens if an Object Is Not Stored in Cache?

If after the backend response Varnish decides that an object will not be stored in cache, it puts the object on a "blacklist"—the so-called *hit-for-pass* cache.

For a duration of 120 seconds, the next requests will immediatly connect with the backend, directly serving the response, without attempting to store the response in cache.

After 120 seconds, upon the next request, the response can be re-evaluated and a decision can be made whether or not to store the object in cache.

How Long Does Varnish Cache an Object?

Once an object is stored in cache, a decision must be made on the time-to-live. I mentioned this before, but there's a list of priorities that Varnish uses to decide which value it will use as the TTL.

Here's the prioritized list:

1. If `beresp.ttl` is set in the `VCL`, use that value as the time-to-live.
2. Look for an `s-maxage` statement in the `Cache-control` header.
3. Look for a `max-age` statement in the `Cache-control` header.
4. Look for an `Expires` header.
5. Cache for 120 seconds under all other circumstances.

When the object is stored in the *hit-for-pass* cache, it is cached for 120 seconds, unless you change the value in *VCL*.

Conclusion

When you're up and running and sending your HTTP traffic through Varnish, there will be a certain behavior that will impact the cacheability of your website.

This behavior does not reflect arbitrary rules and policies that were defined by Varnish itself. Varnish respects conventional HTTP best practices that were defined in industry-wide, accepted RFCs.

Even if you don't add any VCL code, the best practices will make sure that your website is properly cached, assuming that your code respects the best practices as well.

An additional advantage is that the cacheability of your website and the portability of the caching behavior can go beyond the scope of Varnish. You can swap out Varnish for another kind of reverse proxy, or even a *CDN*.

At this point you will know what a `Cache-control` header is and how it compares to an `Expires` header. You'll have a pretty solid idea how to leverage those headers to control the cacheability of your pages. By now, you're no stranger to *Cache variations* and *conditional requests*.

Finally and most importantly: you can only cache `GET` or `HEAD` requests, because they are *idempotent*. *Nonidempotent* requests like, for example, `POST`, `PUT`, and `DELETE` cannot be cached.

CHAPTER 4

The Varnish Configuration Language

As mentioned before, Varnish is a *reverse caching proxy*. There are many other reverse proxies out there that do caching, even in the open source ecosystem. The main reason Varnish is so popular is, without a doubt, Varnish Configuration Language (VCL)—a *domain-specific language* used to control the behavior of Varnish.

The flexibility that VCL offers is unprecedented in this kind of software. It's more a matter of expressing and controlling the behavior by programming it rather than by declaring it in a configuration file. Because of the rich API that is exposed through the objects in VCL, the level of detail with which you can tune Varnish is second to none.

The curly braces, the semicolon statement endings, and the commenting style in VCL remind you of programming languages like C, C++, and Perl. That's maybe why VCL feels so intuitive; it sure beats defining rules in an XML file.

The Varnish Configuration Language doesn't just feel like C, it actually gets compiled to C and dynamically loaded as a shared object when the VCL file is loaded by the Varnish runtime. We can even call it *transpiling*, because we convert a piece of source code to source code in another programming language.

> If you're curious what the C code looks like, just run the `varnishd` program with the `-C` option to see the output.

In this chapter you'll learn how VCL will allow you to hook into the *finite state machine* of Varnish to programmatically extend its behavior. We'll cover the various subroutines, objects, and variables that allow you to extend this behavior.

I already hinted at the built-in VCL in Chapter 3. In this chapter you'll see the actual code of the built-in VCL.

Hooks and Subroutines

VCL is not the kind of language where you start typing away in an empty file or within a main method; it actually restricts you and only allows you to hook into certain aspects of the Varnish execution flow. This execution flow is defined in a *finite state machine*.

The hooks represent specific stages of the Varnish flow. The behavior of Varnish in these stages is expressed through various built-in subroutines. You define a subroutine in your VCL file, extend the caching behavior in that subroutine, and issue a reload of that VCL file to enable that behavior.

Every subroutine has a fixed set of return statements that represent a state change in the flow.

> If you don't explicitly define a return statement, Varnish will fall back on the built-in VCL that is hardcoded in the system. This can potentially undo the extended behavior you defined in your VCL file.
>
> This is a common mistake, one I've made very early on. And mind you: this is actually a good thing because the built-in VCL complies with HTTP best practices.
>
> I actually advise you to minimize the use of custom VCL and rely on the built-in VCL as much as possible.

> **You'll Spend 90% of Your Time in vcl_recv**
>
> When you write VCL, you'll spend about 90% of your time in `vcl_recv`, 9% in `backend_response`, and the remaining 1% in various other subroutines.

Client-Side Subroutines

Here's a list of client-side subroutines:

`vcl_recv`
: Executed at the beginning of each request.

`vcl_pipe`
: Pass the request directly to the backend without caring about caching.

`vcl_pass`
: Pass the request directly to the backend. The result is not stored in cache.

`vcl_hit`
: Called when a cache lookup is successful.

`vcl_miss`
: Called when an object was not found in cache.

`vcl_hash`
: Called after `vcl_recv` to create a hash value for the request. This is used as a key to look up the object in Varnish.

`vcl_purge`
: Called when a purge was executed on an object and that object was successfully evicted from the cache.

`vcl_deliver`
: Executed at the end of a request when the output is returned to the client.

`vcl_synth`
: Return a synthetic object to the client. This object didn't originate from a backend fetch, but was synthetically composed in VCL.

Backend Subroutines

And here's a list of backend subroutines:

`vcl_backend_error_fetch`
: Called before sending a request to the backend server.

`vcl_backend_response`
: Called directly after successfully receiving a response from the backend server.

`vcl_backend_error`
: Executed when a backend fetch was not successful or when the maximum amount of retries has been exceeded.

Initialization and Cleanup Subroutines

And finally there a two subroutines that are used to deal with the initialization and cleanup of VMODs:

`vcl_init`
: Called when the VCL is loaded. VMODs can be initialized here.

`vcl_fini`
: Called when the VCL was executed. VMODs can be cleaned up here.

Custom Subroutines

You can also define your own subroutines and call them from within your VCL code. Custom subroutines can be used to organize and modularize VCL code, mostly in an attempt to reduce code duplication.

The following example consists of a `remove_ga_cookies` subroutine that contains find and replace logic using regular expressions. The end result is the removal of Google Analytics tracking cookies from the incoming request.

Here's the file that contains the custom subroutine:

```
sub remove_ga_cookies {
    # 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.=[^;]+(; )?", "");
}
```

Here's how you call that subroutine:

```
include "custom_subroutines.vcl";

sub vcl_recv {
    call remove_ga_cookies;
}
```

Return Statements

Whereas the VCL subroutines represent the different states of the state machine, the return statement within each subroutine allows for state changes.

If you specify a valid return statement in a subroutine, the corresponding action will be executed and a transition to the corresponding state will happen. As mentioned before: when you don't specify a return statement, the execution of the subroutine will continue and Varnish will fall back on the built-in VCL.

Here's a list of valid return statements:

`hash`
: Look the object up in cache.

`pass`
: Pass the request off to the backend, but don't cache the result.

`pipe`
: Pass the request off to the backend and bypass any caching logic.

`synth`
: Stop the execution and immediately return synthetic output. This returns statement takes an HTTP status code and a message.

`purge`
: Evict the object and its variants from cache. The URL of the request will be used as an identifier.

`fetch`
: Pass the request off to the backend and try to cache the response.

`restart`
: Restart the transaction and increase the `req.restarts` counter until `max_restarts` is reached.

`deliver`
: Send the response back to the client.

`miss`
: Synchronously refresh the object from the backend, despite a hit.

`lookup`
: Use the hash to look an object up in cache.

`abandon`
: Abandon a backend request and return a HTTP 503 (backend unavailable) error.

The execution flow

In Chapter 3, I talked about the built-in VCL and in the previous section I listed a set of subroutines and return statements. It's time to put all the pieces of the puzzle together and compose the execution flow of Varnish.

In Chapter 1, I referred to the finite state machine that Varnish uses. Let's have a look at it and see how Varnish transitions between states and what causes these transitions.

Figure 4-1 shows a simplified flowchart that explains the execution flow.

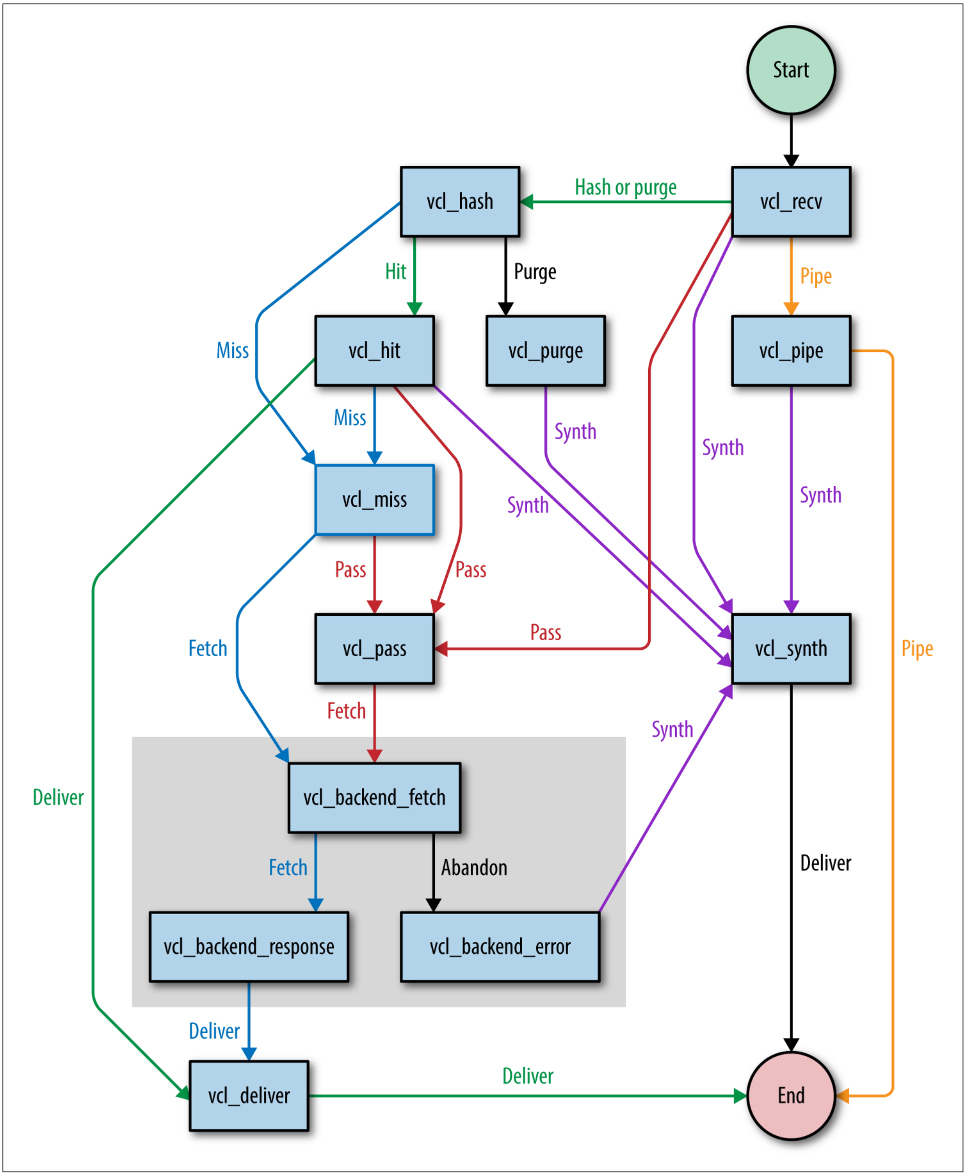


Figure 4-1. The simplified flow of execution in Varnish

We can split up the flow into two parts:

1. Backend fetches (the gray box)
2. Request and response handling (the rest of the flowchart)

The purpose of the split is to handle backend fetches asynchronously. By doing that, Varnish can serve stale data while a new version of the cached object is being fetched. This means less request queuing when the backend is slow.

We have now reached a point where the subroutines start making sense. To summarize, let's repeat some of the important points of the execution flow:

- Every session starts in `vcl_recv`.
- Cache lookups happen in `vcl_hash`.
- Non-cacheable requests are directly passed to the backend in `vcl_pass`. Responses are not cached.
- Items that were found in cache are handled by `vcl_hit`.
- items that were not found are handled by `vcl_miss`.
- Cache misses or passed requests are fetched from the backend via `vcl_backend_fetch`.
- Backend responses are handled by `vcl_backend_response`.
- When a backend fetch fails, the error is handled by `vcl_backend_error`.
- Valid responses that were cached, passed, or missed are delivered by `vcl_deliver`.

At this point you know the basic vocabulary we'll use to refer to the different stages of the finite state machine. Now it's time to learn about the VCL syntax and the VCL objects in order to modify HTTP requests and responses and in order to transition to other stages of the flow.

VCL Syntax

If you want to hook into the Varnish execution flow and extend the subroutines, you'd better know the syntax. Well, let's talk syntax.

Varnish version 4 features a quite significant VCL syntax change compared to version 3: every VCL file should start with `vcl 4.0;`.

Many of the VCL examples in this book do not begin with `vcl 4.0;` because I assume they're just extracts and not the full VCL file. Please keep this in mind.

The full VCL reference manual can be found on the Varnish website.

Operators

VCL has a bunch of operators you can use to assign, compare, and match values.

Here's an example where we combine some operators:

```
sub vcl_recv {
    if(req.method == "PURGE" || req.method == "BAN") {
        return(purge);
    }
    if(req.method != "GET" && req.method != "HEAD") {
        return(pass);
    }
    if(req.url ~ "^/products/[0-9]+/"){
        set req.http.x-type = "product";
    }
}
```

- We use the *assignment operator* (`=`) to assign values to variables or objects.
- We use the *comparison operator* (`==`) to compare values. It returns `true` if both values are equal; otherwise, `false` is returned.
- We use the *match operator* (`~`) to perform a regular expression match. If the value matches the regular expression, `true` is returned; otherwise, `false` is returned.
- The *negation operator* (`!`) returns the inverse logical state.
- The *logical and operator* (`&&`) returns `true` if both operands return true; otherwise, `false` is returned.

In the preceding example, we check if:

- The request method is either equal to `PURGE` or to `BAN`.
- The request method is not equal to `GET` and to `HEAD`.
- The request URL matches a regular expression that looks for product URLs.

There's also the less than operator (`<`), the greater than operator (`>`), the less than or equals operator (`<=`), and the greater than or equals operator (`>=`). Go to the operator section of the Varnish documentation site to learn more.

Conditionals

`if` and `else` statements—you probably know what they do. Let's skip the theory and just go for an example:

```
sub vcl_recv {
    if(req.url == "/"){
        return(pass);
    } elseif(req.url == "/test") {
```

```
        return(synth(200,"Test succeeded"));
    } else {
        return(pass);
    }
}
```

Basically, VCL supports `if`, `else`, and `elseif`. That's it!

Comments

Comments are parts of the VCL that are not interpreted but used to add comments to describe your VCL.

VCL offers three ways to add comments to your VCL:

- Single-line comments using a double slash `//`
- Single-line comments using a hash `#`
- Multiline comments in a comment block that is delimited by `/*` and `*/`

Here's a piece of VCL code that uses all three commenting styles:

```
sub vcl_recv {
    // Single line of out-commented VCL.
    # Another way of commenting out a single line.
    /*
        Multi-line block of commented-out VCL.
    */
}
```

Scalar Values

You can use strings, integers, and booleans—your typical scalar values—in VCL. VCL also supports time and durations.

Let's figure out what we can do with those so-called scalar values.

Strings

Strings are enclosed between double quotes and cannot contain new lines. Double quotes cannot be used either, obviously. If you're planning to use new lines or double quotes in your strings, you'll need to use *long strings* that are enclosed between double quotes and curly braces.

Let's see some code. Here's an example of normal and long strings:

```
sub vcl_recv {
    set req.http.x-test = "testing 123";
    set req.http.x-test-long = {"testing '123', or even "123" for that matter"};
    set req.http.x-test-long-newline = {"testing '123',
or even "123"
```

```
for that matter"};
}
```

Strings are easy—just remember that long strings allow new lines and double quotes, whereas regular strings don't.

Integers

Nothing much to say about integers—they're just numbers. When you use integers in a string context, they get casted to strings.

Here's an example of a valid use of integers:

```
sub vcl_recv {
    return(synth(200,"All good"));
}
```

The first argument of the `synth` function requires an integer, so we gave it an integer.

```
sub vcl_recv {
    return(synth(200,200));
}
```

The preceding example is pretty meaningless; the only thing it does is prove that integers get casted to strings.

Booleans

Booleans are either true or false. Besides a code example, there's nothing more to add:

```
sub vcl_backend_response {
    if(beresp.http.set-cookie) {
        set beresp.uncacheable = true;
    }
}
```

This example does not store the object in cache if the HTTP response contains a `Set-Cookie` header.

Durations

Another type that VCL supports is durations. These are used for timeouts, time-to-live, age, grace, keep, and so on.

A duration looks like a number with a string suffix. The suffix can be any of the following values:

- `ms`: milliseconds

- `s`: seconds
- `m`: minutes
- `h`: hours
- `d`: days
- `w`: weeks
- `y`: years

So if we want the duration to be three weeks, we define the duration as `3w`.

Here's a VCL example where we set the time-to-live of the response to one hour:

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

Durations can contain real numbers. Here's an example in which we cache for 1.5 hours:

```
sub vcl_backend_response {
    set beresp.ttl = 1.5h;
}
```

Regular Expressions

VCL supports Perl Compatible Regular Expressions (PCRE). Regular expressions can be used for pattern matching using the ~ match operator.

Regular expressions can also be used in functions like `regsub` and `regsuball` to match and replace text.

I guess you want to see some code, right? The thing is that I already showed you an example of regular expressions when I talked about the match operator. So I'll copy/paste the same example to prove my point:

```
sub vcl_recv {
    if(req.url ~ "^/products/[0-9]+/"){
        set req.http.x-type = "product";
    }
}
```

Functions

VCL has a set of built-in functions that perform a variety of tasks. These functions are:

- `regsub`
- `regsuball`

- `hash_data`
- `ban`
- `synthetic`

Regsub

`regsub` is a function that matches patterns based on regular expressions and is able to return subsets of these patterns. This function is used to perform find and replace on VCL variables. `regsub` only matches the first occurrence of a pattern.

Here's a real-life example where we look for a language cookie and extract it from the `cookie` header to perform a cache variation:

```
sub vcl_hash {
    if(req.http.Cookie
    ~ "language=(nl|fr|en|de|es)"){
        hash_data(regsub(req.http.Cookie,
        "^.*;? ?language=(nl|fr|en|de|es)( ?|;| ;).*$","\1"));
    }
}
```

> By putting parenthesis around parts of your regular expression, you group these parts. Each group can be addressed in the *sub* part. You address a group by its grade. In case of the previous example, group `\1` represents the first group. That's the group that contains the acutal language we want to extract.

Regsuball

The only difference between `regsub` and `regsuball` is the fact that the latter matches all occurences, whereas the former only matches the first occurence.

When you have to perform a find and replace on a string that has multiple occurences of the pattern you're looking for, `regsuball` is the function you need!

Example? Sure! Check out Example 4-1.

Example 4-1. Remove all Google Analytics cookies by using the `regsuball` function

```
sub vcl_recv {
  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.=[^;]+(; )?", "");
}
```

You might remember this example from "Custom Subroutines" on page 38. If you use Google Analytics, there will be some tracking cookies in your cookie header. These cookies are controlled by Javascript, not by the server. They basically interfere with our hit rate and we want them gone.

The `regsuball` function is going to look for all occurences of these patterns and remove them.

The very first line, will be responsible for removing the following cookies:

- `__utma`
- `__utmb`
- `__utmc`
- `__utmt`
- `__utmv`
- `__utmz`

We really need `regsuball` to do this, because `regsub` would only remove the first cookie that is matched.

Hash_data

The `hash_data` function is used in the `vcl_hash` subroutine and adds data to the hash that is used to identify objects in the cache.

The following example is the same one we used in the `regsub` example: it adds the language cookie to the hash. Because we didn't explicitly mention a return statement, the hostname and the URL will also be added after the execution of `vcl_hash`:

```
sub vcl_hash {
    if(req.http.Cookie ~ "language=(nl|fr|en|de|es)"){
        hash_data(regsub(req.http.Cookie,
        "^.*;? ?language=(nl|fr|en|de|es)( ?|;| ;).*$","\1"));
    }
}
```

Ban

The `ban` function is used to ban objects from the cache. All objects that match a specific pattern are invalidated by the internal ban mechanism of Varnish.

We will go into more detail on banning and purging in Chapter 5. But just for the fun of it, I'll show you an example of a `ban` function:

```
sub vcl_recv {
    if(req.method == "BAN") {
        ban("req.http.host == " + req.http.host + " && req.url == " + req.url);
```

```
        return(synth(200, "Ban added"));
    }
}
```

What this example does is remove objects from the cache when they're called via the *BAN* HTTP method.

> I know, the *BAN* method isn't an official HTTP method. Don't worry, it's only for internal use.
>
> Be sure to put this piece of VCL before any other code that checks for HTTP methods. Otherwise, your request might end up getting piped to the backend. This will also happen if you don't return the synthetic response.

The `ban` function takes a string argument that matches the internal metadata of the cached object with the values that were passed. If an object matches these criteria, it is added to the ban list and removed from cache upon the next request.

Synthetic

The `synthetic` function returns a synthetic HTTP response in which the body is the value of the argument that was passed to this function. The input argument for this function is a string. Both normal and long strings are supported.

Synthetic means that the response is not the result of a backend fetch. The response is 100% artificial and was composed through the `synthetic` function. You can execute the `synthetic` function multiple times and upon each execution the output will be added to the HTTP response body.

The actual status code of such a response is set by `resp.status` in the `vcl_synth` subroutine. The default value is, of course, `200`.

The `synthetic` function is restricted to two subroutines:

- `vcl_synth`
- `vcl_backend_error`

These are the two contexts where no backend response is available and where a synthetic response makes sense.

Here's a code example of synthetic responses:

```
sub vcl_recv {
    return(synth(201,"I created something"));
}

sub vcl_backend_error {
```

```
    set beresp.http.Content-Type = "text/html; charset=utf-8";
    synthetic("An error occured: " + beresp.reason + "<br />");
    synthetic("HTTP status code: " + beresp.status + "<br />");
    return(deliver);
}

sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    synthetic("Message of the day: " + resp.reason + "<br />");
    synthetic("HTTP status code: " + resp.status + "<br />");
    return(deliver);
}
```

Synthetic output doesn't just contain a string of literals. You can also parse input values. As you can see in the preceding examples, we're using the reason and the status to get the body and the HTTP status code.

Mind you, in `vcl_synth` we get these variables through the `resp` object. This means we're directly intercepting it from the response that will eventually be sent to the client.

In `vcl_backend_error`, we don't use the `resp` object, but the `beresp` object. `beresp` means *backend response*. So an attempt has been made to fetch data from the backend, but it failed. Instead, the error message is added to the `beresp.reason` variable.

Includes

Although you will try to rely on the built-in VCL as much as possible, in some cases you'll still end up with lots of VCL code.

After a while, the sheer amount of code makes it hard to maintain the overview. Includes help you organize your code. An `include` statement will be processed by the VCL compiler and it will load the content of included file inline.

In the example below we just load the `someFile.vcl` file. The contents of that file will be placed within the `vcl_recv` subroutine:

```
sub vcl_recv {
    include "someFile.vcl";
}
```

And this is what the `someFile.vcl` file could look like:

```
if ((req.method != "GET" && req.method != "HEAD")
|| req.http.Authorization || req.http.Cookie) {
    return (pass);
}
```

Importing Varnish Modules

Imports allow you to load Varnish modules (*VMODs*). These modules are written in C and they extend the behavior of Varnish and enrich the VCL syntax.

Varnish ships with a couple of VMODs that you can enable by importing them.

Here’s an example where we import the `std` VMOD:

```
import std;
sub vcl_recv {
    set req.url = std.querysort(req.url);
}
```

The example above executes the `querysort` function and returns the URL with a sorted set of querystring parameters. This is important, because when a client uses the querystring parameters in different order, it will trigger a cache miss.

Go to the *`vmod_std`* documentation page to learn all about this very useful VCL extension.

Backends and Health Probes

All the VCL we covered so far has been restricted to the hooks that allow us to change Varnish’s behavior. But let’s not forget that Varnish is all about caching backend responses. So it’s about time we deal with the VCL aspect of backends.

In general, it looks like this:

```
backend name {
    .attribute = "value";
}
```

> Varnish automatically connects to the backend that was defined first. Other backends can only be used by assigning them in VCL using the `req.backend_hint` variable.

Here’ a list of supported backend attributes:

`host`
: A mandatory attribute that represents the hostname or the IP of the backend.

`port`
: The backend port that will be used. The default is port 80.

`connect_timeout`
: The amount of time Varnish waits for a connection with the backend. The default value is 3.5 seconds.

`first_byte_timeout`
: The amount of time Varnish waits for the first byte to be returned from the backend after the initial connection. The default value is 60 seconds.

`between_bytes_timeout`
: The amount of time Varnish waits between each byte to ensure an even flow of data. The default value is 60 seconds.

`max_connections`
: The total amount of simultaneous connections to the backend. When the limit is reached, new connections are dropped. Make sure your backend can handle this amount of connections.

`probe`
: The backend probe that will be used to check the health of the backend. It could be defined inline, or linked to.

Let's throw in a code example:

```
backend default {
    .host = "127.0.0.1";
    .port = "8080";
    .connect_timeout = 2s;
    .first_byte_timeout = 5s;
    .between_bytes_timeout = 1s;
    .max_connections = 150;
}
```

In the preceding example, a connection is made to the local machine on port 8080. We're willing to wait two seconds for an initial connection. Once the connection is established, we're going to wait up to five seconds to get the first byte. After that we want to receive bytes with a regular frequency. We're willing to wait one second between each byte. We will allow up to 150 simultaneous connections to the backend.

> If more requests to the backend are queued than what is allowed by the `max_connections` setting, they will fail.

If any of the criteria is not met, a backend error is thrown and the execution is passed to `vcl_backend_error` with an HTTP 503 status code.

Without the use of a backend probe, an unhealthy backend can only be spotted when the connection fails. A so-called backend probe will poll the backend on a regular basis and check its health based on certain assertions. Probes can be defined similarly to backends.

In general, it looks like this:

```
probe name {
    .attribute = "value";
}
```

Here's a list of supported probe attributes:

`url`
: The url that is requested by the probe. This defaults to `/`.

`request`
: A custom HTTP request that can be sent to the backend.

`expected_response`
: The expected HTTP status code that is returned by the backend. This defaults to 200.

`timeout`
: The amount of time the probe waits for a backend response. The default value is 2 seconds.

`interval`
: How often the probe is run. The default value is 5 seconds.

`window`
: The amount of polls that are examined to determine the backend health. The default value is 8.

`threshold`
: The amount of polls in the *window* that should succeed before a backend is considered healthy. This defaults to 3.

`initial`
: The amount of initial backend polls it takes when Varnish starts before a backend is considered healthy. Defaults to *threshold* - 1.

Here's an example of a backend probe that is defined inline:

```
backend default {
    .host = "127.0.0.1";
    .port = "8080";
    .probe = {
        .url = "/";
        .expected_response = 200;
```

```
        .timeout = 1s;
        .interval = 1s;
        .window = 5;
        .threshold = 3;
        .initial = 2;
    }
}
```

This is what happens: the backend tries to connect to the local host on port 8080. There is a backend probe available that determines the backend health. The probe polls the backend on port 8080 on the root URL. In order for a poll to succeed, an HTTP response with a 200 status code is expected, and this response should happen within a second.

Every second, the probe will poll the backend and the result of five polls is used to determine health. Of those five polls, at least three should succeed before we consider the backend to be healthy. At startup time, this should be two polls.

We can also define a probe explicitly, name it, and reuse that probe for multiple backends. Here's a code example that does that:

```
probe myprobe {
    .url = "/";
    .expected_response = 200;
    .timeout = 1s;
    .interval = 1s;
    .window = 5;
    .threshold = 3;
    .initial = 2;
}

backend default {
    .host = "my.primary.backend.com";
    .probe = myprobe;
}

backend backup {
    .host = "my.backup.backend.com";
    .probe = myprobe;
}
```

This example has two backends, both running on port 80, but on separate machines. We use the `myprobe` probe to check the health of both machine, but we only define the probe once.

In Chapter 6 we'll cover some more advanced backend and probing topics.

Access Control Lists

Access control lists, or ACLs as we like to call them, are language constructs in VCL that contain IP addresses, IP ranges, or hostnames. An ACL is named and IP addresses can be matched to them in VCL.

ACLs are mostly used to restrict access to certain parts of your content or logic based on the IP address. The match can be done by using the match operator (~) in an if-clause.

Here's a code example:

```
acl allowed {
    "localhost";    # myself
    "192.0.2.0"/24; # and everyone on the local network
    ! "192.0.2.23"; # except for one specific IP
}

sub vcl_recv {
    if(!client.ip ~ allowed) {
        return(synth(403,"You are not allowed to access this page."));
    }
}
```

In the preceding example, only local connections are allowed, or connections that come from the 192.0.2.0/24 IP range, with one exception: connections from 192.0.2.23 aren't allowed. No other IP addresses are allowed access, either—they will get an HTTP 403 error if they do try.

In Chapter 5, we'll be using ACLs to restrict access to the cache invalidation mechanism.

VCL Variables

You're probably already familiar with `req.url` and `client.ip`. Yes, these are VCL variables, and let me tell you, there are a lot of them.

Here's a list of the different variable objects:

`bereq`
: The backend request data structure.

`beresp`
: The backend response variable object.

`client`
: The variable object that contains information about the client connection.

`local`
: Information about the local TCP connection.

`now`
: Information about the current time.

`obj`
: The variable object that contains information about an object that is stored in cache.

`remote`
: Information about the remote TCP connection. This is either the client or a proxy that sits in front of Varnish.

`req`
: The request variable object.

`req_top`
: Information about the top-level request in a tree of ESI requests.

`resp`
: The response variable object.

`server`
: Information about the Varnish server.

`storage`
: Information about the storage engine.

Table 4-1 lists a couple of useful variables and explains what they do. For a full list of variables, go to the variables section in the VCL part of the Varnish documentation site.

Table 4-1. A couple of useful VCL objects

Variable	Returns	Meaning	Readable from	Writeable from
beresp.do_esi	boolean	Process the Edge Side Includes after fetching it. Defaults to false. Set it to true to parse the object for ESI directives. Will only be honored if req.esi is true.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.do_stream	boolean	Deliver the object to the client directly without fetching the whole object into varnish. If this request is passed it will not be stored in memory.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.grace	duration	Set to a period to enable grace.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error

Variable	Returns	Meaning	Readable from	Writeable from
beresp.http.	header	The corresponding HTTP header.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.keep	duration	Set to a period to enable conditional backend requests. The keep time is the cache lifetime in addition to the time-to-live.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.reason	string	The HTTP status message returned by the server.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.status	integer	The HTTP status code returned by the server.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.ttl	duration	The object's remaining time-to-live, in seconds.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
beresp.uncacheable	boolean	Setting this variable makes the object uncacheable, which may get stored as a hit-for-pass object in the cache.	vcl_backend_response, vcl_backend_error	vcl_backend_response, vcl_backend_error
client.identity	string	Identification of the client, used to load balance in the client director. Defaults to *`client.ip`*.	client	client
client.ip	IP	The client's IP address.	client	
local.ip	IP	The IP address of the local end of the TCP connection.	client	
obj.age	duration	The age of the object.	vcl_hit	
obj.grace	duration	The object's remaining grace period in seconds.	vcl_hit	
obj.hits	integer	The count of cache-hits on this object. A value of 0 indicates a cache miss.	vcl_hit, vcl_deliver	
obj.ttl	duration	The object's remaining time-to-live, in seconds.	vcl_hit	
remote.ip	IP	The IP address of the other end of the TCP connection. This can either be the client's IP or the outgoing IP of a proxy server.	client	
req.backend_hint	backend	Sets bereq.backend to this value when a backend fetch is required.	client	client
req.hash_always_miss	boolean	Force a cache miss for this request. If set to true, Varnish will disregard any existing objects and always (re)fetch from the backend. This allows you to update the value of an object without having to purge or ban it.	vcl_recv	vcl_recv
req.http.	header	The corresponding HTTP header.	client	client

Variable	Returns	Meaning	Readable from	Writeable from
req.method	string	The request type (e.g., GET, HEAD, POST, …).	client	client
req.url	string	The requested URL.	client	client
resp.reason	string	The HTTP status message returned.	vcl_deliver, vcl_synth	vcl_deliver, vcl_synth
resp.status	int	The HTTP status code returned.	vcl_deliver, vcl_synth	vcl_deliver, vcl_synth
server.ip	IP	The IP address of the server on which the client connection was received.	client	

Varnish's Built-In VCL

Remember "Varnish Built-In VCL Behavior" on page 31 in which I talked about the built-in VCL behavior of Varnish? Now that we know how VCL works, we can translate that behavior into a full-blown VCL file.

Even if you don't register a VCL file or if your VCL file only contains a backend definition, Varnish will behave as follows:[1]

```
/*-
 * Copyright (c) 2006 Verdens Gang AS
 * Copyright (c) 2006-2015 Varnish Software AS
 * All rights reserved.
 *
 * Author: Poul-Henning Kamp <phk@phk.freebsd.dk>
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
```

1 This piece of VCL code is the VCL that is shipped when you install Varnish.

```
 * SUCH DAMAGE.
 *

 *
 * The built-in (previously called default) VCL code.
 *
 * NB! You do NOT need to copy & paste all of these functions into your
 * own vcl code, if you do not provide a definition of one of these
 * functions, the compiler will automatically fall back to the default
 * code from this file.
 *
 * This code will be prefixed with a backend declaration built from the
 * -b argument.
 */

vcl 4.0;

#######################################################################
# Client side

sub vcl_recv {
    if (req.method == "PRI") {
        /* We do not support SPDY or HTTP/2.0 */
        return (synth(405));
    }
    if (req.method != "GET" &&
      req.method != "HEAD" &&
      req.method != "PUT" &&
      req.method != "POST" &&
      req.method != "TRACE" &&
      req.method != "OPTIONS" &&
      req.method != "DELETE") {
        /* 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);
}

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);
}

sub vcl_pass {
    return (fetch);
}

sub vcl_hash {
    hash_data(req.url);
    if (req.http.host) {
        hash_data(req.http.host);
    } else {
        hash_data(server.ip);
    }
    return (lookup);
}

sub vcl_purge {
    return (synth(200, "Purged"));
}

sub vcl_hit {
    if (obj.ttl >= 0s) {
        // A pure unadultered 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);
}

sub vcl_miss {
    return (fetch);
}

sub vcl_deliver {
    return (deliver);
}

/*
 * We can come here "invisibly" with the following errors: 413, 417 & 503
 */
sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";
    synthetic( {"<!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);
}

#######################################################################
# Backend Fetch

sub vcl_backend_fetch {
    return (fetch);
}

sub vcl_backend_response {
    if (beresp.ttl <= 0s ||
      beresp.http.Set-Cookie ||
      beresp.http.Surrogate-control ~ "no-store" ||
      (!beresp.http.Surrogate-Control &&
        beresp.http.Cache-Control ~ "no-cache|no-store|private") ||
      beresp.http.Vary == "*") {
        /*
        * Mark as "Hit-For-Pass" for the next 2 minutes
        */
        set beresp.ttl = 120s;
        set beresp.uncacheable = true;
    }
    return (deliver);
}

sub vcl_backend_error {
    set beresp.http.Content-Type = "text/html; charset=utf-8";
    set beresp.http.Retry-After = "5";
    synthetic( {"<!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);
}

#######################################################################
# Housekeeping

sub vcl_init {
}

sub vcl_fini {
    return (ok);
}
```

As a quick reminder, this is what the preceding code does:

- It does not support the `PRI` method and throws an `HTTP 405` error when it is used.
- Request methods that differ from `GET`, `HEAD`, `PUT`, `POST`, `TRACE`, `OPTIONS`, and `DELETE` are not considered valid and are piped directly to the backend.
- Only `GET` and `HEAD` requests can be cached, other requests are passed to the backend and will not be served from cache.
- When a request contains a cookie or an authorization header, the request is passed to the backend and the response is not cached.
- If at this point the request is not passed to the backend, it is considered cacheable and a cache lookup key is composed.
- A cache lookup key is a hash that is composed using the URL and the hostname or IP address of the request.
- Objects that aren't stale are served from cache.
- Stale objects that still have some grace time are also served from cache.
- All other objects trigger a miss and are looked up in cache.
- Backend responses that do not have a positive TTL are deemed uncacheable and are stored in the hit-for-pass cache.
- Backend responses that send a `Set-Cookie` header are also considered uncacheable and are stored in the hit-for-pass cache.
- Backend responses with a no-store in the `Surrogate-Control` header will not be stored in cache either.

- Backend responses containing no-cache, no-store, or private in the `Cache-control` header will not be stored in cache.
- Backend responses that have a `Vary` header that creates cache variations on every request header are not considered cacheable.
- When objects are stored in the hit-for-pass cache, they remain in that blacklist for 120 seconds.
- All other responses are delivered and stored in cache.

A Real-World VCL File

The VCL file that you see in "Varnish's Built-In VCL" on page 57 represents the desired behavior of Varnish. In an ideal world, this VCL code should suffice to make any website bulletproof. The reality is that modern day websites, applications and APIs don't conform 100% to these rules.

The built-in VCL assumes that cacheable websites do not have cookies. Let me tell you: websites without cookies are few and far between. The following VCL file deals with these real-world cases and will dramatically increase your hit rate.

> Although this real-world VCL file increases your hit rate, it is not tuned for any specific CMS or framework. If you happen to need a VCL file that caters to these specific applications, it will usually come with a CMS or framework module.

There are plenty of good VCL templates out there. I could also write one myself, but I'd just be reinventing the wheel. In the spirit of open source, I'd much rather showcase one of the most popular VCL templates out there.

The author of the VCL file is Mattias Geniar, a fellow Belgian, a fellow member of the hosting industry, a friend, and a true Varnish ambassador. Go to his GitHub repository to see the code.

This VCL template primarily sanitizes the request, optimizes backend connections, facilitates purges, adds caching stats, and adds ESI support.

> Room for improvements? Just send Mattias a pull request and explain why.

Conclusion

By now you should know the syntax, functions, different language constructs, return types, variable objects, and execution flow of VCL. Use this chapter as a reference when in doubt.

Don't forget that the Varnish Cache project has a pretty decent documentation site. If you don't find the answer to your question in this book, you'll probably find it there.

At this point, I expect you to be comfortable with the VCL syntax and be able to read and interpret pieces of VCL you come across. In Chapter 7, we'll dive deeper into some common scenarios in which custom VCL is required.

CHAPTER 5

Invalidating the Cache

In this chapter I'll highlight several cache invalidation strategies. These strategies allow you to remove certain items from cache even though their time-to-live hasn't expired yet.

In the world of caching, there's only one thing worse than a low hit rate, and that's caching for too long. That statement sounds quite weird, right? Here I am trying to convince you to cache everything, all the time, yet I'm saying that caching for too long is the worst thing to do. Allow me to explain.

Caching for Too Long

Throughout this book, I've always kept the best interests of the website owner, developer, and sysadmin in mind. The reality is that the site, API, and application are primarily services that the end user consumes. In the end, it's all about the end user.

- Why do we want to make the site fast? For the user!
- Why do we want to keep the site available? For the user!
- Why do we cache? So that the user has a good experience!

Caching data for too long would mess with the integrity of the data, giving the user a bad experience when up-to-date output is important. This is especially the case for news websites.

We already talked about the use of `Cache-control` and `Expires` headers. It's important to estimate the right time-to-live and set the right values for these headers. The more accurate the time-to-live, the better the balance.

Unfortunately, in many cases the data will be out-of-date even before the object expires. Setting it to a lower value could jeopardize the health and responsiveness of your backend. Talk about being stuck between a rock and a hard place!

Worry not—Varnish has your back! Varnish offers various mechanisms to evict objects from the cache based on certain criteria. By accessing these eviction mechanisms from your code, you can actively invalidate objects, even if they haven't expired. That way your breaking news will be correctly displayed on the front page of your website, even if the object still has two hours to live according to the time-to-live.

The Varnish documentation has a page dedicated to cache invalidation. Have a look if you're interested.

Purging

Purging is the easiest way to invalidate the cache. In the following example, you can see that in VCL you can perform a `return (purge)` from within the `vcl_recv` subroutine. This will explicitly evict the object from cache. The object will be identified by the criteria set in `vcl_hash`, so by default that is the hostname and the URL. Memory is freed up immediately and cache variations are also evicted.

```
acl purge {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    # allow PURGE from localhost and 192.168.55...

    if (req.method == "PURGE") {
        if (!client.ip ~ purge) {
            return(synth(403,"Not allowed."));
        }
        return (purge);
    }
}
```

There's a bit more housekeeping involved when you want to do it right: the preceding example protects you from unauthorized invalidations by enforcing an ACL. Only purges from `localhost` or from the `192.168.55.0/24` subnet are allowed.

And then there's the `PURGE` request method that is checked. By requesting the resource with `PURGE` instead of `GET`, you're basically telling Varnish that this HTTP request is not a regular data retrieval request, but a purging request.

> You probably remember "When Does Varnish Completely Bypass the Cache?" on page 31. In that section, I mentioned that only certain request methods will be considered valid by Varnish. `PURGE` is not one of them. That's why it's important to do the `PURGE` check before the request method validation happens. Otherwise, you'll go into `pipe` mode and the request will be sent to the backend either returning a valid HTTP 200 status code or if your webserver doesn't allow `PURGE`, an HTTP 405 error.

You can implement a purge call anywhere in your code and you'll typically use an HTTP client that is supported by your programming language or framework. In many cases, that client will be cURL-based. Here's a purging example using the cURL binary:

```
curl -XPURGE http://example.com/some/page
```

This example uses the `-X` parameter in cURL to set the request method. As expected, we're setting it to `PURGE` and setting the URL to *http://example.com/some/page*. That's the resource we're removing from cache.

Banning

Purging is easy: it uses the object's hash, it evicts just that one object, and it can be executed with a simple `return(purge)`.

But when you have a large number of purges to perform or you're not exactly sure which resources are stale, exact URL invalidations might feel restrictive to you. A *pattern-based invalidation mechanism* would solve that problem, and banning does just that.

Banning should not be an unknown concept to you; in "Ban" on page 47, we talked about the `ban` function that executes these bans.

Basically, bans use a regular expression match to mark objects that should be removed from cache. These marked objects are put on the so-called ban list. Banning does not remove items from cache immediately and hence does not free up any memory directly.

Bans are checked when an object is hit and executed accordingly based on the ban list. There's also a so-called ban lurker background thread that checks for bans that match against any variable of the `obj` object.

The `obj` object only stores the response headers, response body, and metadata. It has no request information. The *ban lurker* doesn't have any of this information either, which is why the *ban lurker thread* can only remove items from cache if the ban matches objects that have no request context, like `obj`.

All other bans are removed at request time and aren't done in the background.

Here's a basic `BAN` example. It does exactly the same thing as the `PURGE` example, but adds URL pattern-matching capabilities:

```
acl ban {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.method == "BAN") {
        if (!client.ip ~ ban) {
            return(synth(403, "Not allowed."));
        }
        ban("req.http.host == " + req.http.host +
            " && req.url ~ " + req.url);
        return(synth(200, "Ban added"));
    }
}
```

When you accumulate lots of bans based on `req` object variables for resources that are not frequently accessed, Varnish might run into CPU performance problems.

Bans are kept on the ban list until all objects in cache have been checked against the list. If the banned objects do not get a new hit, they remain on the list. The longer the list, the more CPU time is required to check the list upon every hit.

That's why it's advised to use lurker-friendly bans.

Lurker-Friendly Bans

The *ban lurker* is in charge of asynchronously checking and cleaning up the ban list. I mentioned that the ban lurker has a limited scope to invalidate objects because of its lack of request information: the ban lurker only knows the `obj` context.

But if we copy request information from the `req` object, we can actually write lurker-friendly bans. Have a look at the following VCL snippet:

```
acl ban {
    "localhost";
```

```
    "192.168.55.0"/24;
}

sub vcl_backend_response {
  set beresp.http.x-host = bereq.http.host;
  set beresp.http.x-url = bereq.url;
}

sub vcl_deliver {
  unset resp.http.x-host;
  unset resp.http.x-url;
}

sub vcl_recv {
    if (req.method == "BAN") {
        if (!client.ip ~ ban) {
            return(synth(403, "Not allowed."));
        }
        ban("obj.http.x-host == " + req.http.host +
            " && obj.http.x-url ~ " + req.url);
        return(synth(200, "Ban added"));
    }
}
```

The trick is to add the host and the URL of the request as a response header when the object is stored in cache. By doing this, the missing request context is actually there. I know—it's trickery, but it does the job.

`set beresp.http.x-host = bereq.http.host;` will set a custom `x-host` header containing the host of the request, and `set beresp.http.x-url = bereq.url;` will set the URL as a custom `x-url` response header.

At this point, the invalidation will not just happen at request time on the next hit, but also asynchronously by the ban lurker. The ban lurker will have the necessary request information to process bans on the ban list that contain a URL match.

After the ban we'll just remove these custom headers; because these are for internal purposes, the user has no business using them. That is done in `vcl_deliver`.

The ban lurker doesn't remove items from the ban list immediately; there are three parameters that influence its behavior:

`ban_lurker_age`
: Bans have to be at least this old until they are removed by the ban lurker. The default value is *60 seconds*.

`ban_lurker_batch`
: The number of bans the ban lurker processes during a single run. The default value is *1000 items*.

`ban_lurker_sleep`
: The amount of seconds the ban lurker sleeps in between runs. The default value is *0.010 seconds*.

If you write lurker-friendly bans and your ban list is still long, you might want to take a look at these parameters and tune them accordingly.

More Flexibility

Let's have one more ban example that puts it all together and gives you even more flexibility:

```
acl ban {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_backend_response {
  set beresp.http.x-host = bereq.http.host;
  set beresp.http.x-url = bereq.url;
}

sub vcl_deliver {
  unset resp.http.x-host;
  unset resp.http.x-url;
}

sub vcl_recv {
    if (req.method == "BAN") {
        if (!client.ip ~ ban) {
            return(synth(403, "Not allowed."));
        }
        if(req.http.x-ban-regex) {
            ban("obj.http.x-host == " + req.http.host + "
            && obj.http.x-url ~ " + req.http.x-ban-regex);
        } else {
            ban("obj.http.x-host == " + req.http.host + "
            && obj.http.x-url == " + req.url);
        }
        return(synth(200, "Ban added"));
    }
}
```

This example combines the benefits of the previous ban examples. It gives you the flexibility to choose between an exact URL match or a regular expression match. If you set the `x-ban-regex` request header when banning, the value will be used to match the URL pattern. If the header is not set, the URL itself (and nothing more) is banned. And this, of course, is a lurker-friendly ban.

Here's an example using the cURL binary:

```
curl -XBAN http://example.com/ -H"x-ban-regex: ^/product/[0-9]+/details"
```

In this example, we're purging all product details pages based on the `^/product/[0-9]+/details` regular expression. If you only want to purge a single product detail page, the curl call could look like this:

```
curl -XBAN http://example.com/product/121/details
```

> The name of the request method we're using to ban or purge doesn't really matter. As long as you can identify an invalidation request, you're fine. We're just calling it `BAN` or `PURGE`. Choose a request method name of your liking—just make sure it doesn't clash with another method you use in your backend application.

Viewing the Ban List

If you're interested in seeing the current state of the ban list, you can issue a `ban.list` command on the `varnishadm` administration program. Just execute the following command on your Varnish server:

```
varnishadm ban.list
```

And this could be the output:

```
Present bans:
0xb75096d0 1318329475.377475    10      obj.http.x-host == example.com
&& obj.http.x-url ~ ^/product/[0-9]+/details
0xb7509610 1318329470.785875    20C     obj.http.x-host == example.com
&& obj.http.x-url ~ ^/category
```

Wondering what each field means? Here we go:

- The first field contains the unique identifier of the ban.
- The second field is the timestamp.
- The third field represents the amount of objects in cache that match this ban. Optionally, there could be a `C` attached to the third field—this is a completed ban match, usually for duplicate bans.
- The fourth field is the ban expression itself.

Banning from the Command Line

The previous two sections approached the execution of cache invalidation from an HTTP perspective, meaning that your regular requests and your purge/ban requests all pass through the same channel. I mentioned the upside: it's very easy to code in VCL and just as easy to implement in your backend.

There are also some downsides:

- You have to write additional VCL code, which adds complexity.
- There is no uniform way of implementing banning and purging in Varnish; your application will depend on the invalidation implementation in VCL.
- Although the ACLs provide a level of security, there is no isolation from a networking perspective.

Luckily, Varnish offers an admin console that allows you to issue ban statements. This can be done locally or remotely through the `varnishadm` program. In "CLI address binding" on page 13, I mentioned how you can configure your Varnish to accept remote connections on the admin interface.

`varnishadm` is just a client that connects to the Varnish CLI socket. You can also make a TCP connection to this socket and issue ban commands directly from within your code. Have a look at the documentation page on the Varnish Command Line Interface to learn more about the commands, remote connections, and the authentication protocol.

Here's an example of our product detail invalidation, but this time using the CLI:

```
varnishadm> ban obj.http.x-host == example.com && obj.http.x-url ~^/product/↵
[0-9]+/details
200
```

The preceding command should be typed as one line without a linebreak. The break in the line is there strictly because of the page constraints.

Forcing a Cache Miss

The responsibility of `ban` and `purge` is to remove items from the cache—not storing the new data in cache. The next user that consumes the invalidated resource triggers the re-entry of this object in cache. This is an asynchronous process and there might be some time between the invalidation request and the actual re-entry of the data in cache.

For a synchronous "reload" of the data, `req.hash_always_miss` is a better solution. By setting `req.hash_always_miss` to true in `vcl_recv`, you're telling Varnish to get the latest value for that resource, even if the object in cache hasn't expired. A new object is set, containing the up-to-date value. The old object is no longer used, but will still be there until it expires.

`req.hash_always_miss` doesn't free up memory. On the contrary, it adds extra objects to cache instead.

You can use this refresh strategy, for example, if you're an editor of a website. You use an ACL to identify yourself and your requests will never return a cached version of the resource. Every time you update that resource, you immediatly see the updated version. The added value to the end user is that they also see the up-to-date version of that resource, but the result gets cached.

Here's an example that illustrates the use of `req.hash_always_miss`:

```
acl editors {
    "localhost";
    "192.168.55.0"/24;
}

sub vcl_recv {
    if (req.http.cache-control ~ "no-cache" && client.ip ~ editors) {
         set req.hash_always_miss = true;
    }
}
```

The cherry on the cake here is that this example only forces a miss when the client does a force refresh in the browser.

Cache Invalidation Is Hard

Cache invalidation is important, but not always easy to implement. With *purge*, *ban*, and *force refresh*, Varnish makes the job easier. Unfortunately, these are just instruments and tools. The real difficulty is implementing these mechanism in your application.

The invalidations can be built as scheduled maintenance jobs, or they could be hooks in your content management system. So far, so good—but knowing what resources to invalidate can sometimes be quite hard.

Imagine an ecommerce site that sells products. Products can be part of multiple categories. You can have featured products on your homepage. If you decide to update the product name, you need to invalidate the following resources:

- The product detail page
- The homepage
- All the category pages where the product appears
- Maybe even an API or a feed where this product appears

- Who knows, maybe you have a multilingual shop that requires invalidations accross multiple languages

Phil Karlton was right:

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> —Phil Karlton

Proper cache invalidation requires a lot of insight into the application and the way HTTP is used. If you're using frameworks like Wordpress, Drupal, or Magento, you'll find third-party modules that handle the invalidation for you.

A final piece of advice I can give you regarding cache invalidation is the use of *surrogate keys*. Instead of purging URLs or URL patterns, you can use surrogate keys to register tags under the form of an HTTP response header called `xkey`; it's just a matter of invalidating pages by tag instead of URL.

To perform tag-based invalidation, you can use the `vmod_xkey` that is part of the Varnish modules package. I won't cover surrogate keys in detail because it's beyond the scope of this book. Have a look at the page, install the modules package, and go ahead and try it yourself. This can be very useful when lots of related invalidations need to happen. Invalidating by tag seems like a nice approach.

Let's cover yet another approach: remember "Conditional Requests" on page 25? If your backend is highly optimized for these conditional requests, there is no need to purge or ban. Conditional requests will automatically update the cache if the backend data has changed and will return an HTTP 304 as long as the data remains unmodified.

> The only way to make conditional request invalidation work reliably is if your backend code can handle `If-Modified-Since` or `If-None-match` requests without suffering from an increase in load. To avoid stale data, you will typically have low times-to-live to benefit from conditional requests, which puts extra pressure on your backend application.

Conclusion

At this point, you know that invalidating your cache is important and that Varnish offers banning, purging, and force-refresh to perform these invalidations.

You can either invalidate the cache using HTTP calls, or you can connect to `varnishadm` and issue the `ban` command there without having to write specific VCL to provide invalidation.

Cache invalidation is not just important—it's also really hard. The trick is that you need to be able to map your HTTP routes to the impacted data.

CHAPTER 6

Dealing with Backends

Although the strategy is to minimize the use of the backend, it is still a crucial part of the equation. Without the backend, we cannot cache any objects, and preferably it's the backend that decides what gets cached and how long objects are stored there.

In this chapter, we'll talk about backends and how you can configure access to them using VCL. We'll also group backends and perform load balancing using directors. Dealing with healthy and unhealthy backends will also be covered in this chapter.

Backend Selection

In previous chapters, I mentioned that there are two ways to announce the backend to Varnish:

- By adding a backend to your VCL file.
- Or by omitting VCL completely and using the `-b` flag at startup time.

That's how you do it automatically. But there are situations where there are multiple backends and you want to control which request goes to which backend. You can define multiple backends and use `req.backend_hint` to assign a backend other than the default one.

Here's an example:

```
vcl 4.0;

backend public {
  .host = "web001.example.com";
}

backend admin {
```

```
    .host = "web002.example.com";
}

sub vcl_recv {
    if(req.url ~ "^/admin(/.*)?") {
        set req.backend_hint = admin;
    } else {
        set req.backend_hint = public;
    }
}
```

You notice that we defined two backends:

- A backend that handles the public traffic and that resides on *web001.example.com*.
- A backend that serves the admin panel and that resides on *web002.example.com*.

By incorporating the `req.backend_hint` variable in our VCL logic, we can perform content-aware load balancing. Each backend could be tuned to its specific task.

> Yes, Varnish has load-balancing capabilities. However, I don't consider Varnish a true load balancer. To me, HAProxy is a superior open source load balancer, whereas Varnish is a superior HTTP accelerator. If you don't need some of the more advanced features that HAProxy offers, Varnish will do the job just fine.

Backend Health

In "Backends and Health Probes" on page 50, we discussed how health checks can be performed using health probes. Let's take our previous example and add a health probe:

```
vcl 4.0;

probe healthcheck {
   .url = "/";
   .interval = 5s;
   .timeout = 1s;
   .window = 10;
   .threshold = 3;
   .initial = 1;
   .expected_response = 200;
}

backend public {
  .host = "web001.example.com";
  .probe = healthcheck;
}
```

```
backend admin {
  .host = "web002.example.com";
  .probe = healthcheck;
}

sub vcl_recv {
    if(req.url ~ "^/admin(/.*)?") {
        set req.backend_hint = admin;
    } else {
        set req.backend_hint = public;
    }
}
```

The health checks in this example are done based on an HTTP request to the homepage. This check is done every five seconds with a timeout of one second. We expect an HTTP 200 status code. The backends are only considered healthy if 3 out of 10 health checks succeed.

You can view the health of your backends by executing the following command:

```
varnishadm backend.list
```

This could be the output of that command:

```
Backend name                   Admin      Probe
boot.public                    probe      Healthy 10/10
boot.admin                     probe      Healthy 10/10
```

Both backends are listed and their health is automatically checked by a probe (see the Admin column). It is considered healthy because 10 out of 10 checks were successful. As a quick reminder: only three need to succeed to have a healthy backend.

To get more verbose output, use the `-p` parameter:

```
varnishadm backend.list -p
```

This could be the output:

```
Backend name                   Admin      Probe
boot.public                    probe      Healthy 10/10
  Current states  good: 10 threshold:  3 window: 10
  Average response time of good probes: 0.575951
  Oldest ================================================== Newest
  4444444444444444444444444444444444444444444444444444444444444444 Good IPv4
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX Good Xmit
  RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR Good Recv
  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH Happy

boot.admin                     probe      Healthy 3/10
  Current states  good:  3 threshold:  3 window: 10
  Average response time of good probes: 0.642713
  Oldest ================================================== Newest
  4444444444444444444444444444444444444444444444---------------444 Good IPv4
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX---------------XXX Good Xmit
```

```
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR---------------RRR Good Recv
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH---------------HHH Happy
```

You see that the admin backend's last three checks were successful, and those three are enough to be considered healthy. This is a backend that recovered from being "sick."

If you continuously run this command (the `watch` command on Linux can help you with that), you can see the evolution of the health checks (`Xmit`), the responses (`Recv`), and the *happiness* of your server.

You can also manually override the probe's decision on the backend health by issuing the following commands:

```
varnishadm backend.set_health boot.public sick
varnishadm backend.set_health boot.admin healthy
```

The output is the following:

```
Backend name                   Admin      Probe
boot.public                    sick       Healthy 10/10
boot.admin                     healthy    Healthy 10/10
```

As you can see, the `Admin` column no longer contains the value `probe`, but `sick` and `healthy`. To give the control back to the health probes, simply issue the following commands:

```
varnishadm backend.set_health boot.public auto
varnishadm backend.set_health boot.admin auto
```

The `std.healthy()` function that is part of the `vmod_std` can tell you whether or not a backend is healthy. To check the health of the current backend, just use `std.healthy(req.backend_hint)`.

Directors

A director is a built-in VMOD that groups several backends and presents them as one. Directors offer different decision-making strategies to decide which request is handled by which backend.

The goal of directors is to avoid backend latency by distributing requests across multiple backends: by balancing out backend requests, backend servers will experience less load, which reduces latency. This is basically a *horizontal scalability* strategy.

Besides pure load balancing, directors also make sure that unhealthy backend nodes are not used; instead, another healthy node is used to handle those requests. This is essentially a *high availability* strategy.

In order to use directors, you first need to import the `directors` VMOD and initialize it in `vcl_init`. Here's an example:

```
vcl 4.0;
import directors;

backend web001 {
  .host = "web001.example.com";
  .probe = healthcheck;
}

backend web002 {
  .host = "web002.example.com";
  .probe = healthcheck;
}

sub vcl_init {
    new loadbalancing = directors.round_robin();
    loadbalancing.add_backend(web001);
    loadbalancing.add_backend(web002);
}

sub vcl_recv {
    set req.backend_hint = loadbalancing.backend();
}
```

Let's take this example step by step:

1. First we import the `directors` VMOD.
2. We declare two backends.
3. We initialize the director and decide on the load-balancing strategy (in this case, it's "round robin").
4. We assign both backends to the director.
5. We now have a new backend that we named `loadbalancing`.
6. We assign the `loadbalancing` backend to Varnish by issuing `set req.backend_hint = loadbalancing.backend();`.

More information about directors can be found on the Varnish documentation page about vmod_directors.

The Round-Robin Director

The previous example contained an example that referred to the round-robin director, but what does that mean? *Round-robin* is a basic distribution strategy where every backend takes its turn sequentially. If you have three backends, the sequence is:

- Backend 1
- Backend 2
- Backend 3

- Backend 1
- Backend 2
- Backend 3
- …

The load is distributed equally—no surprises whatsoever. In most cases, round-robin is a good algorithm, but there are scenarios where it won't perform well. Take, for example, a situation where your backend servers don't have the same server resources. The server with the least amount of memory or CPU will still have to do an equal amount of work.

Here's an example of a round-robin director declaration that uses three backends:

```
sub vcl_init {
    new loadbalancing = directors.round_robin();
    loadbalancing.add_backend(backend1);
    loadbalancing.add_backend(backend2);
    loadbalancing.add_backend(backend3);
}
```

After we declare the director, it needs to be assigned in `vcl_recv`:

```
sub vcl_recv {
    set req.backend_hint = loadbalancing.backend();
}
```

The Random Director

The *random director* distributes load over the backends using a weighted random-probability distribution algorithm. By default, the weight for all backends is the same, which means load will be (somewhat) equally distributed. In that respect, the random director has the same effect as the round-robin director, with some slight deviations.

As soon as you start assigning specific weights to the backends, the deviations will increase. This makes sense if you want to "spare" one or more servers, such as in a situation where a server is under-dimensioned or hosts other business-critical applications.

Based on the weights, each backend receives `100 × (weight / (sum(all_added_weights)))` percent of the total load.

Here's an example:

```
sub vcl_init {
    new loadbalancing = directors.random();
    loadbalancing.add_backend(backend1,1.0);
    loadbalancing.add_backend(backend2,2.0);
}
```

The preceding example declares a new random director with two backends:

- Backend 1 receives about 33% of the load
- Backend 2 receives about 67% of the load

And then you assign the backend:

```
sub vcl_recv {
    set req.backend_hint = loadbalancing.backend();
}
```

The Hash Director

The *hash director* chooses the backend based on a SHA256 hash of a given value. The value it hashes is passed to the `backend()` method of the director object.

The hash director is often used to facilitate sticky sessions by hashing either the client IP address or a session cookie. By hashing either of those values, Varnish assures that requests for the same user or session always reach the same backend. This is important for backend servers that store their session data locally.

You could also hash by request URL. Requests for the URL will always be sent to the same backend.

The risk of hashing by client IP or request URL is that the load will not be equally distributed, such as in the following cases:

- The client IP address could be the IP of a proxy server used by multiple users, causing heavy load on one specific backend.
- One URL could be far more popular than another, causing heavy load on one specific backend.

This is how you declare a hash director:

```
sub vcl_init {
    new loadbalancing = directors.hash();
    loadbalancing.add_backend(backend1,1.0);
    loadbalancing.add_backend(backend2,1.0);
}
```

This example assigns two backends with the same weight, which is the recommended value. If you change the weights, one server will get more requests than the other, but requests for the same user, URL, or session will continuously be sent to the same backend.

With hash directors, the magic is in the assignment, not the declaration. Here's an example of a hash director that hashes by client IP address:

```
sub vcl_recv {
    set req.backend_hint = loadbalancing.backend(client.ip);
}
```

As I explained, sticky-IP hashing is risky. Here's an example in which the session cookie is hashed:

```
sub vcl_recv {
    set req.backend_hint = loadbalancing.backend(regsuball(req.http.Cookie,
    "^.*;? ?PHPSESSID=([a-zA-Z0-9]+)( ?|;| ;).*$","\1"));
}
```

This example uses the `PHPSESSID` cookie that contains the session ID generated by the `session_start()` function in PHP.

And here's a final example, in which we do URL hashing:

```
sub vcl_recv {
    set req.backend_hint = loadbalancing.backend(req.url);
}
```

The Fallback Director

The final director I'm going to feature is the *fallback director*. For this director, the order of backend assignments is very important: the fallback director will try each backend and return the first one that is healthy.

> Please ensure that your backends have a health probe attached; otherwise, the fallback director has no way to determine whether or not backends are healthy.
>
> Without the presence of a health probe, the fallback director will not try the next backend and will return an HTTP 503 error.

Let's have a look at an example of the fallback director:

```
vcl 4.0;

import directors;

probe healthcheck {
   .url = "/";
   .interval = 2s;
   .timeout = 1s;
   .window = 3;
   .threshold = 2;
   .initial = 1;
   .expected_response = 200;
}

backend web001 {
```

```
    .host = "web001.example.com";
    .probe = healthcheck;
}

backend web002 {
    .host = "web002.example.com";
    .probe = healthcheck;
}

sub vcl_init {
    new loadbalance = directors.fallback();
    loadbalance.add_backend(web001);
    loadbalance.add_backend(web002);
}

sub vcl_recv {
    set req.backend_hint = loadbalance.backend();
}
```

In this example, web001 is the preferred backend. The health probe will check the availability of the homepage every two seconds. If two out of three checks succeed, the backend is considered healthy; otherwise, the fallback director will switch to web002.

> It is possible to stack directors and to use directors as members of other directors. This is a way to combine loadbalancing strategies.
>
> For example, you can have round-robin directors that each have two backends in two data centers. To ensure round-robin load balancing and still have high-availability, you can add both round-robin directors as members of a fallback director.

Grace Mode

Throughout this chapter, we've focused on providing a stable backend service so that Varnish can access backend data without the slightest hiccup:

- Measuring backend health
- Adding health check probes
- Offering directors to distribute load
- Leveraging directors to use healthy nodes instead of unhealthy ones

But one important question remains unanswered: what do we do if no backends are available?

We can either live with it or compensate for it. I'd rather go for the latter, and that's where *grace mode* comes into play.

If we assign a certain amount of grace time, we're basically telling Varnish that it can serve objects beyond their time-to-live. These objects are considered "stale" and are served as long as there's no updated object for a duration defined by the grace time.

The built-in VCL hit logic explains it best:

```
sub vcl_hit {
    if (obj.ttl >= 0s) {
        // A pure unadultered 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 the object hasn't expired (obj.ttl >= 0s), keep on serving the object
- If the object has expired, but there is some grace time left (obj.ttl + obj.grace > 0s), keep serving the object but fetch a new version asynchronously
- Otherwise, fetch a new version and queue the request

It's important to get a major misconception out of the way: *grace mode* only works for items that are stored in cache. It's the only way to make the reload go unnoticed. As long as the object is in cache, it can be served and the backend call is asynchronous.

But when the requested object is not stored in cache, the backend request is synchronous and the end user has to wait for the result, even though the backend fetch is done by a separate thread.

Enabling Grace Mode

Enabling grace mode is quite easy: you just assign a value to `beresp.grace` in the `vcl_backend_response` subroutine:

```
sub vcl_backend_response {
    set beresp.grace = 30s;
}
```

In this example, we're allowing Varnish to keep serving stale data up to 30 seconds beyond the object's time-to-live. When that time expires, we revert back to synchronous fetching. This means that slow backends will feel slow again and if the backend is down, an HTTP 503 error will be returned.

Conclusion

You've spent a lot of time learning how Varnish works and how to get data in the cache. A lot of attention has gone to the *client-side* aspect: serving cached data to the end user. But let's not forget that Varnish relies on a healthy backend to do its work.

At this point, you know how to link Varnish to one or multiple backends. You can configure all kinds of timeouts, and you're able to check the health of a backend. You should also feel confident circumventing unhealthy backends with directors or by using grace mode.

By using some of the tips and tricks from this chapter, you'll have more uptime. Regardless of your hit rate, you will still need your backend for content revalidation, so be sure to think about the availability of your backend servers.

CHAPTER 7

Improving Your Hit Rate

Up to this point, you've learned about best practices for implementing Varnish under ideal circumstances. But the real world is often less than ideal, and you need to know what to do when things don't go according to plan.

This chapter will strike a balance between optimizing your application and just writing VCL to cope with the limitations of your application.

Common Mistakes

I remember when I started out with Varnish, I thought I had it all figured out. I read the manual, copy-and-pasted some VCL, and added a layer of Varnish to a client's website. All of this was an attempt to protect my client's website from the impact of a huge marketing campaign.

But it didn't go as planned. I was caching too aggressively and even cached the cookies. This resulted in a cached version of the shopping cart and login page. Safe to say that the client wasn't happy.

There were other instances where the hit rate was spectacularly low and that was because I was unaware of the built-in VCL. I was just writing VCL in `vcl_recv` without exiting the subroutine with return statements like `hash` or `pass`. Varnish just went about its business and continued using default behavior that discarded my changes.

I see clients who write their own VCL make similar mistakes. Long story short: there are a bunch of common mistakes that people make, and here's how to avoid the most common pitfalls.

Not Knowing What Hit-for-Pass Is

Many people forget that Varnish creates a *hit-for-pass* object when fetched objects cannot be cached. Varnish will cache the decision not to cache for 120 seconds. Objects in the hit-for-pass cache will not be queued like regular misses and will directly make a backend connection. This mechanism has an impact and can lead to strange conclusions if you don't prepare some contingencies.

Imagine a situation where you set `s-maxage` temporarily to zero for quick debugging purposes. When you reset the value to a cacheable value within 120 seconds of the previous request, it still won't be cached. This confuses people.

My advice

Purge or ban the URL that is stuck in hit-for-pass and the updated `Cache-control` header will kick in.

Returning Too Soon

Another very common mistake is forgetting about the consequences of return statements. Most people know that Varnish doesn't cache POST requests and they assume that the Varnish engine will deal with that.

So they write a VCL snippet like this to cache everything besides the admin pages:

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

But in this example, there is no way Varnish can fall back on the built-in VCL—there were no measures in place to handle POST requests. So in this case, all request methods can be cached.

> Be aware that if you're trying to cache POST calls or calls with other non-idempotent request methods, the initial request to the backend will be transformed into a GET and the request body will be discarded.

My advice

Either include code from the built-in VCL that you need, or just remove the explicit `return(hash);` to fall back on built-in VCL.

Purging Without Purge Logic

When people hear that you can purge URLs from Varnish by using the `PURGE` method, they tend to get excited and forget that purge support needs to be implemented in your VCL file.

I've seen setups where people use `PURGE` without it being implemented and without people even noticing it. Even the default VCL will not complain when you issue a `PURGE`: you'll hit the if statement that checks the request methods, and your request will be piped to the backend and you'll probably get an HTTP 200 back.

Another variation on this problem is implementing `PURGE` support after the request method check. Even though you implemented it, your request will still be piped.

My advice

Check for the proper use of `PURGE` and `BAN`. To avoid shenanigans like this, you could also issue ban statements using the command-line interface, as explained in "Banning from the Command Line" on page 71.

No-Purge ACL

Cache invalidation using the `PURGE` method and the `return(purge)` logic is very simple to implement. However, if you don't protect access to your purge logic via an ACL, you can end up in major trouble. Any other script kiddie could scan your site map and execute `PURGE` calls on every URL on your site. All caching would then go out of the window and the load on your backend servers would dramatically increase.

My advice

Add an ACL, obviously! Have a look at "Purging" on page 66 for an example. If you want to tighten the security of your `PURGE/BAN` logic, just use the command-line interface and firewall access to the interface. Again, read "Banning from the Command Line" on page 71 for more information.

404 Responses Get Cached

Remember that time you deleted a file, and then quickly re-uploaded it so that no one would notice? Well, they sure did notice it, because while you were uploading, someone accessed that page and now the 404 response is cached.

Varnish caches responses that have the following status codes:

- *200*: OK
- *203*: Non-Authoritative Information

- *300*: Multiple Choices
- *301*: Moved Permanently
- *302*: Moved Temporarily
- *307*: Temporary Redirect
- *410*: Gone
- *404*: Not Found

All other status codes aren't cached.

Want to make sure 404s aren't cached? Use the following VCL snippet:

```
sub vcl_backend_response {
    if (beresp.status == 404) {
        set beresp.ttl = 0s;
        return(deliver);
    }
}
```

Setting an Age Header

In "Expiration" on page 22, I mentioned how expiration works in HTTP. I also added a word of warning, saying that Varnish uses the `Age` header to determine how long it still has to cache an object.

If you start setting your own `Age` headers, it will mess with the cache duration and that might become a problem.

My advice

Don't set an `Age` header yourself; let Varnish do that for you.

Max-age Versus s-maxage

Most people in the web industry have heard about `max-age`. A lot fewer people know what `s-maxage` does. Having both mechanisms in your `Cache-control` header could lead to confusion, especially if the values differ.

My advice

Always set `s-maxage` and if you want the browser to do some extra caching, you can use `max-age`. Use the instruments that HTTP provides and use them in the right way. When in doubt, read Chapter 3.

Adding Basic Authentication for Acceptance Environments

Imagine this situation—I assume it will sound familiar: you're working on a new project for a client. It's almost done and you want the client to test the latest features. The production environment is ready to go, but to avoid spoiling details to the outside world too early, you added server-side basic authentication.

During those tests, you notice that the hit rate drops dramatically. And yes, that makes sense, because built-in VCL doesn't not cache requests that include an `Authorization` header.

My advice

Use firewall rules or VPN access instead. If you really want basic authentication, you could handle the authentication on a node in front of Varnish. If you terminate SSL/TLS, that server could act as the authentication server.

You could also use `vmod_basicauth` and handle the authentication at the Varnish level instead of the backend level.

Here's a code example:

```
vcl 4.0;

import basicauth;

sub vcl_recv {
    if (!basicauth.match("/var/www/.htpasswd",req.http.Authorization)) {
        return(synth(401,"Authentication required"));
    }
    unset req.http.Authorization;
}
```

Session Cookies Everywhere

One of my pet peeves is the use of session cookies "just in case." Those are the kind of cookies that are generated in the very first step of your application flow—because you never know, right?

My advice

Minimize the use of cookies. Use session cookies in places where it matters. Other meta information, such as the language of the user, can be stored in separate cookies.

Dedicated cookies make more sense and are easier to control by Varnish. If these cookies get in the way of your hit rate, you can use URL blacklists or whitelists to decide where cookies should be used and where they should be removed.

No Cache Variations

A typical case where a lack of cache variations poses a problem is a multilingual site that uses the `Accept-Language` header to determine the language and render output in the appropriate language.

If there is no cache variation on that header, the language of the first request will be the cached language. As a developer, you should be aware of this and add a `Vary: Accept-Language` header to your responses.

The same problem occurs when you don't perform a cache variation on the `Accept-Encoding` header: compressed content that offers no support for compression could be served to the client.

My advice

Be aware of cache variations, and know how and when to use the `Vary` header.

Do You Really Want to Cache Static Assets?

In many cases, you have way more files to cache than available storage space in Varnish. When Varnish runs out of available storage, it uses a Least Recently Used (LRU) strategy to free up space by removing items with the least amount of hits.

Unfortunately, you don't have any control over what that actually is. But let's be honest: most of your object storage will be consumed by static assets like images, videos, PDF documents, and other similar files. We call them *static assets* because they don't change and because their output isn't rendered by a runtime. They are just files that are transmitted by the web server.

In most cases, it's not the images, JavaScript files, or even the CSS files that cause load on the backend—it's the dynamic scripts. Scripts written in languages like PHP, Ruby, Python, ASP.NET, and many more. Scripts that depend on external resources like databases, feeds, or web services. Scripts that are rendered by a runtime and that consume a bunch of RAM and CPU.

These are the situations where caching matters and where Varnish can make a difference. These scripts are tiny and consume little space in the object storage of Varnish. The number of dynamic scripts on a machine isn't that big and is controlled by the deployment.

Images and documents, however, are often controlled by the users: they can upload images and documents using a CMS. It's tough to control the number of (somewhat) large files uploaded by end users.

So instead of taking our chances an trusting that the LRU strategy will do a good job, we can also decide not to cache static assets in Varnish. The only reason we would cache static assets in Varnish is to avoid consuming too many backend connections.

A web server like Nginx is built to handle these things like a champ! High concurrency? No problem.

If you believe that images and other large files are filling up your cache like crazy, then use the following piece of VCL:

```
vcl 4.0;

sub vcl_recv {
    if (req.url ~ "^[^?]*\.(7z|avi|bmp|bz2|css|csv|doc|docx|eot|flac|flv|gif|gz
    |ico|jpeg|jpg|js|less|mka|mkv|mov|mp3|mp4|mpeg|mpg|odt|otf|ogg|ogm|opus|pdf
    |png|ppt|pptx|rar|rtf|svg|svgz|swf|tar|tbz|tgz|ttf|txt|txz|wav|webm|webp
    |woff|woff2|xls|xlsx|xml|xz|zip)(\?.*)?$") {
        return (pass);
    }
}
```

You could also provide a separate hostname for these static files and bypass Varnish completely—for example, *http://static.example.com/image.jpg*.

URL Blacklists and Whitelists

Controlling what does and doesn't get cached is often done with URL blacklists and whitelists. This means you match URLs or URL patterns and decide whether or not to cache them.

Here's an example of a *blacklist*. It's part of a VCL template for Drupal 7:

```
sub vcl_recv {
    if (req.url ~ "^/status\.php$" ||
        req.url ~ "^/update\.php$" ||
        req.url ~ "^/ooyala/ping$" ||
        req.url ~ "^/admin"        ||
        req.url ~ "^/admin/.*$"    ||
        req.url ~ "^/user"         ||
        req.url ~ "^/user/.*$"     ||
        req.url ~ "^/users/.*$"    ||
        req.url ~ "^/info/.*$"     ||
        req.url ~ "^/flag/.*$"     ||
        req.url ~ "^.*/ajax/.*$"   ||
        req.url ~ "^.*/ahah/.*$")
    {
        return (pass);
    }
}
```

The blacklist in this example typically passes requests to the backend where the URL represents parts of Drupal that contain user-specific data. That's also a place where session cookies are used to identify the user.

You could also switch it up and go for a *whitelist* instead:

```
sub vcl_recv {
    if (req.url ~ "^/products(/.*)?$")
    {
        return (hash);
    }
}
```

In this example, we're caching the `/products` page, but also all subordinate pages like `/products/cellphones`.

Decide What Gets Cached with Cache-Control Headers

In the previous section, blacklists and whitelists were used to decide whether or not to cache. Though effective, it's extra VCL code and hence not that portable; I'm all about empowering the application and its developers.

A better way to do this is by letting the application send a `no-store` for resources that shouldn't be cached. This gives developers more control. Cache behavior changes do not require VCL changes in that case.

Here's a bit of code to illustrate this fact:

```
Cache-control: private, no-cache, no-store
```

By sending these headers, Varnish will not cache the response and add the request to the hit-for-pass cache.

There is one major downside, though: the decision happens at the response level, not at the request level. This makes it a lot harder to deal with cookies.

Here's an unconventional way to still make it work, even when there are cookies:

```
sub vcl_recv {
    if (req.method != "GET" && req.method != "HEAD") {
        return (pass);
    }
    return(hash);
}
```

Basically, you're caching everything that is a `GET` or a `HEAD`, even if there are cookies involved. There's a risk of caching pages that require cookie values, but that's the responsibility of the developer. All pages that require cookie interaction should have a `no-store` response header.

Although I'm saying that `no-store` is the way to go, you can achieve the same result by using one of the following `Cache-control` instructions:

- `no-cache`
- `private`
- `s-maxage=0`
- `max-age=0`

There Will Always Be Cookies

I don't like cookies. They annoy me. But we need them because they make a stateless protocol like HTTP stateful. And that's what we sometimes need: to be able to pass extra information about the user across requests. By default, Varnish doesn't cache when there's a `Cookie` header, and in an ideal setup, we would avoid the use of cookies.

The reality is that there will always be cookies, even if you don't use any in your code. Simple fact: as soon as you start using Google Analytics, you'll have tracking cookies. And that's just one example of tracking cookies.

And let me tell you: there is not a single solution to this problem. It all depends on the type of cookie and its role within the application. Instead of a long conceptual explanation, I'll show you some use cases and code examples.

Admin Panel

Imagine a website that is powered by a CMS. The pages themselves are pretty static, but the admin panel hosted under */admin* is dynamic and requires a login. Once you're logged in, there's a session cookie that keeps track of your session ID. Sound familiar?

This is the VCL you need to cache the website itself, but still have dynamic access to the admin panel:

```
sub vcl_recv {
    if (!(req.url ~ "^/admin/")) {
        unset req.http.Cookie;
    }
}
```

Since we only need cookies when we're in the /admin section, we can strip them off for all other pages. Notice that we're not "returning"; we're relying on built-in VCL for the rest of the flow.

Remove Tracking Cookies

Tracking cookies have a bad rep: people think they're used to "spy" on us. But marketers rely on them to retrieve user metrics.

Let's face it: if you run a website, you're going to install Google Analytics and you're going to want to know how many daily visitors you have, what kind of content they are consuming, and how they found your site. Guess what: this requires tracking cookies. Even if your website is built out of plain old HTML, there will still be cookies.

But these cookies are not processed by the web server or the application runtime; the browser processes them using Javascript. And quite honestly, they're in our way and we want to get rid of them.

The good news is that you can. By stripping off tracking cookies, Varnish will start caching your site again. The Javascript code that processes the cookies is run in the browser. By the time Varnish strips off the cookies, they will already have been processed by the browser.

The following code is a no-brainer and is an extract out of "A Real-World VCL File" on page 62:

```
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, "__gads.=[^;]+(; )?", "");
  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 example performs regex-based find-and-replace magic. It identifies specific cookies and removes them from the cookie header. This is just a set of common tracking cookies; you can extend the list and remove even more cookies.

If after the find-and-replace actions, the cookie header is nothing more than a set of tabs or spaces, you can completely remove the cookie header.

> Please note that vmod_cookie is a Varnish module that offers functions to clean up, delete, and filter cookies without having to write complicated regular expressions. This VMOD is not shipped by default and is part of the Varnish modules collection. If you don't have the possibility of installing this VMOD, the preceding example using regular expressions will do the job just fine.

Remove All But Some

You can also apply the inverse logic: instead of removing some cookies, remove them all except the ones you really need. This makes things easier because you don't have to edit your VCL code every time a new cookie needs to be removed.

Here's the code that only keeps the lang and the PHPSESSID cookie. All the others are removed:

```
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, ";(lang|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;
    }
}
```

Cookie Variations

A very common scenario is using cookies to keep track of user preferences. Language is probably the most common one. The danger of caching those pages is that the language doesn't vary for the cached page. Not caching the page because of the language cookie might be a missed opportunity.

If you have tight control over your language cookie, you can actually do VCL-based cache variations based on that cookie.

Let's have a look at some code:

```
sub vcl_hash {
    if(req.http.Cookie ~ "language=(nl|fr|en|de|es)"){
        hash_data(regsub(req.http.Cookie,
        "^.*;? ?language=(nl|fr|en|de|es)( ?|;| ;).*$","\1"));
    }
}
```

We already featured this code in Chapter 4, but it's a good example and an excellent use case.

And don't forget that the language selection happens on a splash screen. Once a language selection has been made, it just displays the site in the selected language. This splash screen is only shown if no language cookie is set. That means if no cookie is set, we cannot cache.

I know, it sounds like the exact opposite of what I've been preaching, but in this case it's required. The following code is necessary to make this work:

```
sub vcl_recv {
    if(!req.http.Cookie ~ "language=(nl|fr|en|de|es)"){
        return(pass);
    }
}
```

Sanitizing

So far, the only input cleanup we've done was cookie-based. But the way the URL is used can also mess with our hit rate. Here are a couple of quick fixes to optimize your URLs to avoid unnecessary cache misses.

Removing the Port

The following example removes the port number from the HTTP host. The port number might be important from a TCP perspective, but once you've reached Varnish, you're already in. Keeping the port number could mean a cache variation and would cause a miss.

By removing the port, the connectvity aspect is not disrupted, but the host header is sanitized and the hit rate is maintained.

```
sub vcl_recv {
  set req.http.Host = regsub(req.http.Host, ":[0-9]+", "");
}
```

In most cases, the port will not be mentioned in the URL. But by adding port 80 to a regular URL, you could actually cause a cache miss.

Query String Sorting

"Never trust the end user." That point was proven in the previous section where users messed with the hit rate by adding a port. The query string is also vulnerable.

In the following example, we're applying the `std.querysort` function to alphabetically sort query-string parameters. If users put the query-string parameters in a different order, we'd have a different URL, a different hash, and a cache miss as a result.

By sorting the query-string parameters alphabetically, we're eliminating that risk:

```
import std;

sub vcl_recv {
  set req.url = std.querysort(req.url);
}
```

Removing Google Analytics URL Parameters

Google Analytics doesn't just add tracking cookies to your requests; it can also inject tracking URL parameters. It's a common marketing trick to do campaign tracking.

Just like the tracking cookies, these parameters are processed by the browser, serve no purpose to the server, and can be removed.

Here's some code to remove the tracking URL parameters:

```
sub vcl_recv {
  if (req.url ~ "(\?|&)(utm_source|utm_medium|utm_campaign|utm_content|gclid|cx
  |ie|cof|siteurl)=") {
    set req.url = regsuball(req.url, "&(utm_source|utm_medium|utm_campaign
    |utm_content|gclid|cx|ie|cof
    |siteurl)=([A-z0-9_\-\.%25]+)", "");
    set req.url = regsuball(req.url, "\?(utm_source|utm_medium|utm_campaign
    |utm_content|gclid|cx|ie|cof
    |siteurl)=([A-z0-9_\-\.%25]+)", "?");
    set req.url = regsub(req.url, "\?&", "?");
    set req.url = regsub(req.url, "\?$", "");
  }
}
```

Removing the URL Hash

You're probably familiar with the # sign in the URL: it is used as an HTML anchor to link to a specific section of a page. This is a client-side thing, not a server-side thing. We can strip it as follows:

```
sub vcl_recv {
  if (req.url ~ "\#") {
    set req.url = regsub(req.url, "\#.*$", "");
  }
}
```

Removing the Trailing Question Mark

In a URL, the question mark is used to indicate the start of the URL parameters. If the question mark is at the end of your URL, you really don't have any URL parameters. So why keep it?

Here's how you can remove a trailing question mark:

```
sub vcl_recv {
  if (req.url ~ "\?$") {
    set req.url = regsub(req.url, "\?$", "");
  }
}
```

Hit/Miss Marker

A useful addition to your response headers is a custom header that lets you know if the page you're seeing is the result of a cache hit or cache miss. This is the VCL code you need the display the very convenient hit/miss marker:

```
sub vcl_deliver {
  if (obj.hits > 0) {
    set resp.http.X-Cache = "HIT";
  } else {
    set resp.http.X-Cache = "MISS";
  }
}
```

If you consider this hit/miss marker to be sensitive information, you can put an ACL on it and only return the header if the client IP matches the ACL.

You can even go one step further and display the number of hits:

```
sub vcl_deliver {
  set resp.http.X-Hits = obj.hits;
```

```
  if (obj.hits > 0)
    set resp.http.X-Cache = "HIT";
  } else {
    set resp.http.X-Cache = "MISS";
  }
}
```

> You always have the `Age` header to know how long the page has been served from cache so far.

Caching Blocks

In your code, the different sections of your content might be constructed and displayed as blocks. The image below contains one of the most traditional site layouts containing four basic blocks:

- Header
- Footer
- Navigation
- Main block

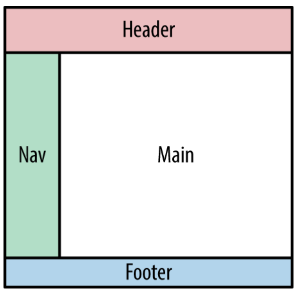


Figure 7-1. The typical block setup with a header, footer, navigation, and main block

If your header section is not cacheable, the rest of the page won't be either. Varnish caches pages and has no understanding of the content blocks. The lowest common denominator of these blocks is a miss, so the page will not be cached.

Should you abandon all hope? Not really. There are ways to circumvent these limitations:

- AJAX

- Edge Side Includes (ESI)

AJAX

AJAX stands for Asynchronous JavaScript and XML and has nothing to do with Varnish. By using AJAX, you're basically letting your browser load the different blocks as separate HTTP requests. This is done in JavaScript, so it's a client-side matter.

The downside is that there's an HTTP request for every content block, which causes overhead. The upside of separate HTTP requests is that you can control the time-to-live and the cacheability of each block by setting the right `Cache-control` headers.

You can use AJAX natively by initializing an `XMLHttpRequest` object, but most of us just use the AJAX implementation of our favorite JavaScript framework.

In terms of caching, it would be great for the application to have full control. Repetition, repetition, repetition: `Cache-control` headers are all you need to make that happen. But if you want to be on "Easy Street" and not care about `Cache-control` discipline, you could just write some VCL to handle it. Here's the idea: if you're only using AJAX to load blocks that would otherwise not be cached, you can pass all AJAX requests to the backend. This is the VCL code that does that for you:

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

You might think that my assumptions are simple and risky, and I agree: when the URL contains an `/ajax/` part, we assume it's AJAX and we don't cache. But hey, it gets the job done!

Edge Side Includes

AJAX is asynchronous, is loaded by the browser, and depends entirely on JavaScript. Although it solves the block-caching problem quite well, the server loses control over the actual output and how it gets rendered. This might not be an acceptable solution to you.

If that's the case, *Edge Side Includes* (or ESI, as we tend to call them) can be a viable alternative. The goal of ESI is to include HTTP fragments in your output that originate from different URLs. The composition of the output happens on *the edge*, rather than on the web server. Every fragment can use `Cache-control` headers to control its own cacheability.

In our case, *the edge* is Varnish and it represents the outer edge of our web stack. ESI was not invented by or for Varnish. It's a standard that is also used by content delivery networks (CDNs), but Varnish only supports a subset of its features.

These so-called fragments are loaded by using an ESI tag, as illustrated below:

```
<esi:include src="http://example.com/include.php" />
```

This tag is nothing more than a placeholder of which the `src` attribute is processed by Varnish. It's very similar to old-school Server-Side Includes (SSI) from back in the day, except that SSI tags were processed by the webserver. ESI tags are processed on the edge. It's like a frameset processed server-side but without the ugliness.

The following HTML code is an example of a page where the main content is loaded by the web server—and the header, navigation, and footer are loaded on the edge:

```
<!DOCTYPE html>
<html>
<head>
    <title>My ESI placeholder</title>
</head>
<body>
    <header>
        <esi:include src="http://example.com/header.php" />
    </header>
    <nav>
        <esi:include src="http://example.com/nav.php" />
    </nav>
    <main>
        <!--This is where the main content will be loaded-->
    </main>
    <footer>
        <esi:include src="http://example.com/footer.php" />
    </footer>
</body>
</html>
```

Making Varnish Parse ESI

Varnish does not automatically parse ESI tags, as this would be a bit resource-intensive. In your VCL code, you explicitly need to enable ESI parsing on a request basis.

The `beresp.do_esi` variable is used to enable and disable ESI parsing.

There are several ways to implement this. Here's an example where a URL prefix is used to enable ESI parsing:

```
sub vcl_backend_response {
    if(bereq.url ~ "^/esi/.*$") {
        set beresp.do_esi=true;
```

```
    }
}
```

In this example, every URL that starts with `/esi/` is expected to have ESI tags in its content that need parsing. But as I have evangelized many, many times before, it's better to let the application take control of the situation.

The following example features ESI parsing based on a response header sent by the application:

```
sub vcl_backend_response {
    if(beresp.http.x-parse-esi) {
        set beresp.do_esi=true;
    }
}
```

But there's a slightly more official way do it. Here's the code I advise you to use:

```
sub vcl_recv {
    set req.http.Surrogate-Capability="key=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;
        return(deliver);
    }
}
```

In this example, we're sending a `Surrogate-Capability` request header to the backend, indicating that we're able to parse ESI. The backend can go ahead and send us ESI tags, knowing we'll be able to handle it.

Once the backend response returns to Varnish, the `vcl_backend_response` subroutine will check if a `Surrogate-Control` response header is sent containing `ESI/1.0`. By sending that header, the application announces that it is returning ESI tags that Varnish is supposed to parse.

For more information about these headers, please read the W3C Edge Architecture Specification.

> If you send `Surrogate-Control` headers from your application, please only do this on the placeholder page that contains your ESI tags. It's not necessary to do this within the resources that are loaded through ESI.

ESI versus AJAX

I made a case for both ESI and AJAX. Which one should you pick?

Personally, I prefer ESI because you don't depend on the browser—or JavaScript for that matter. But ESI has some limitations:

- By default, Varnish requires HTTP fragments loaded by ESI to be XHTML-compliant. Any deviation will cause an error.
- ESI tags are parsed sequentially. If you have a bunch of tags to parse and the corresponding pages are rather slow, this could cause a major slowdown.
- The ESI implementation in Varnish does not offer graceful degradation, which means that Varnish will not recover from a potential error in an ESI subrequest.

The ESI implementation of Varnish also has some advantages:

- You can have nested ESI tags. The depth is limited by the `max_esi_depth` variable. The default maximum depth is set to five levels.
- The `req_top` object gives you the ability to get request information from the top-level request from within an ESI subrequest.
- You can check the level of ESI nesting by using the `req.esi_level` variable.

The XHTML compliance limitation can be circumvented by setting a `feature=esi_disable_xml_check` feature flag at startup time. We talked about this in "Runtime parameters" on page 15.

AJAX, on the other hand, has a couple of clear advantages:

- For many web developers, AJAX is common practice and thus easier to implement
- No real VCL magic required
- AJAX calls can be executed in parallel
- When an AJAX call fails, there is a sense of *graceful degradation*: the failed component can be ignored gracefully and the rest of the page can still be displayed

One of the disadvantages of AJAX is that these requests are sent from the client—and the responses are parsed by the client, too. If you're on an unreliable connection, you'll end up with multiple HTTP calls that can slow down your website. Mobile traffic is a case in point: if you're on a crappy 3G or 2G connection, HTTP round trips are very expensive. If you use ESI instead, all the subrequests will be done server-side and a single round-trip will suffice.

Choose wisely!

Making Your Code Block-Cache Ready

I know I've said it so many times that it might sound annoying by now, but I'm a very strong advocate of empowering the application to make the right decisions in terms of caching. You can bet this also applies to block caching!

Your application might not always be positioned behind a reverse caching proxy, let alone one that supports ESI. I'd advise you to detect if the `Surrogate-Capability` is set before outputting ESI tags for your blocks. If it turns out that this header is not set, I wouldn't take any risks and would just render those blocks via AJAX. There's even a third option: render the blocks natively and don't apply any block caching logic.

Many frameworks have *view helpers* that automatically create ESI tags or AJAX logic. They're usually intelligent enough to parse the subrequest internally if the `Surrogate-Capability` header is not set. By using so-called *view helpers* for this, you're restricting the context to the view layer and you don't have to add extra logic to your MVC controllers. This small widget takes care of the subrequests and makes life easy for you. And in the end, your model and controller layer don't care how the output is visualized.

The next section contains a code example of HTTP fragments loaded through ESI and AJAX. Check it out.

An All-in-One Code Example

Let's end this chapter with an all-in-one code example that applies all these best practices. The code is written in PHP and uses the Silex framework, which is a microframework based on Symfony. The framework is very HTTP-friendly and offers most of the best practices right out of the box.

The example code—available on GitHub—uses the following HTTP best practices:

- Use `Cache-control` headers to decide what does and does not get cached
- Use the `s-maxage` directive to define how long Varnish can cache
- Apply a language cache variation using the `Vary` header
- Use a mixture of ESI tags and AJAX calls to have a separate time-to-live on each content block
- Use view helpers to render the content blocks
- Provide conditional request support for returning ETags and checking the `If-None-Match` header

This is the VCL code you need to make this example work. As you can see, it's pretty limited:

```
vcl 4.0;

sub vcl_recv {
    set req.http.Surrogate-Capability="key=ESI/1.0";
    if ((req.method != "GET" && req.method != "HEAD") || req.http.Authorization) {
        return (pass);
    }
    return(hash);
}

sub vcl_backend_response {
    if(beresp.http.Surrogate-Control~"ESI/1.0") {
        unset beresp.http.Surrogate-Control;
        set beresp.do_esi=true;
        return(deliver);
    }
}
```

The VCL code processes ESI and tries to cache everything that is a `GET` or a `HEAD` request. It's up to the application to decide whether or not this is fine. If not, the object will end up in the hit-for-pass cache.

> Before you can use the example code, you need to load a set of dependencies. These dependencies are managed by Composer.
>
> You must declare your dependencies in a *composer.json* file and run the `composer install` command to load them.

The Composer file

This is the *composer.json* file you need in order to run the example:

```
{
    "require": {
        "silex/silex": "^2.0",
        "twig/twig": "^1.27",
        "symfony/twig-bridge": "^3.1",
        "symfony/translation": "^3.1"
    }
}
```

The PHP code

This is the PHP code that renders the output:

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Symfony\Component\HttpFoundation\Response;
```

```
use Symfony\Component\HttpFoundation\Request;
use Silex\Provider\HttpFragmentServiceProvider;
use Silex\Provider\HttpCacheServiceProvider;

$app = new Silex\Application();
$app['debug'] = true;

$app->register(new Silex\Provider\TwigServiceProvider(),
['twig.path' => __DIR__.'/../views']);
$app->register(new HttpFragmentServiceProvider());
$app->register(new HttpCacheServiceProvider());
$app->register(new Silex\Provider\TranslationServiceProvider(),
['locale_fallbacks' => ['en']]);

$app['locale'] = 'en';
$app['translator.domains'] = [
    'messages' => [
        'en' => [
            'welcome'     => 'Welcome to the site',
            'rendered'    => 'Rendered at %date%',
            'example'     => 'An example page'
        ],
        'nl' => [
            'welcome'     => 'Welkom op de site',
            'rendered'    => 'Samengesteld op %date%',
            'example'     => 'Een voorbeeldpagina'
        ]
    ]
];

$app->before(function (Request $request) use ($app){
    $app['translator']->setLocale($request->getPreferredLanguage());
});

$app->after(function(Request $request, Response $response) use ($app){
    $response
        ->setVary('Accept-Language')
        ->setETag(md5($response->getContent()))
        ->isNotModified($request);
});

$app->get('/', function () use($app) {
    $response =  new Response($app['twig']->render('index.twig'),200);
    $response
        ->setSharedMaxAge(5)
        ->setPublic();
    return $response;
})->bind('home');

$app->get('/header', function () use($app) {
    $response =  new Response($app['twig']->render('header.twig'),200);
    $response
```

```
        ->setPrivate()
        ->setSharedMaxAge(0);
    return $response;
})->bind('header');

$app->get('/footer', function () use($app) {
    $response =  new Response($app['twig']->render('footer.twig'),200);
    $response
        ->setSharedMaxAge(10)
        ->setPublic();
    return $response;
})->bind('footer');

$app->get('/nav', function () use($app) {
    $response =  new Response($app['twig']->render('nav.twig'),200);
    $response
        ->setSharedMaxAge(20)
        ->setPublic();
    return $response;
})->bind('nav');

$app->run();
```

Let's explain what it does:

1. Initialize the autoloader.
2. Resolve the namespaces.
3. Intialize the Silex Application.
4. Register the Twig service provider to load and parse Twig templates.
5. Register the `HttpFragmentServiceProvider` to support ESI and AJAX content block loading.
6. Register the `HttpCacheServiceProvider` to support the ESI view helper.
7. Register the `TranslationServiceProvider` to support translations and leverage the translations view helper.
8. Set English as the default locale.
9. Initialize the translations.
10. Set a pre-dispatch hook that uses the `Accept-Language` header value as the locale.
11. Set a post-dispatch hook that sets the `Vary: Accept-Language` header.
12. Set a post-dispatch hook that sets the `ETag` header.
13. Set a post-dispatch hook that checks the `If-None-Match` header and returns an `HTTP 304` status code if nothing changed.
14. Register a default MVC route that handles the homepage, parses the `index.twig` template, and caches the response for five seconds.

15. Register an MVC route for `/header` that loads the header as a separate HTTP fragment. This resource is not cached.
16. Register an MVC route for `/footer` that loads the footer as a separate HTTP fragment. This resource is cached for 10 seconds.
17. Register an MVC route for `/nav` that loads the navigation as a separate HTTP fragment. This resource is cached for 20 seconds.

All of this in 77 lines of code. Not bad!

The index template

Now we need a template that contains the output for the homepage. Here goes:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/html"
xmlns:hx="http://purl.org/NET/hinclude">
<head>
    <title>Varnish</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css
    /bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on
    3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
    <script src="//rawgit.com/mnot/hinclude/master/hinclude.js"></script>
</head>
<body>
<div class="container-fluid">
    {{ render_hinclude(url('header')) }}
    <div class="row">
        <div class="col-sm-3 col-lg-2">
            {{ render_esi(url('nav')) }}
        </div>
        <div class="col-sm-9 col-lg-10">
            <div class="page-header">
                <h1>{{ 'example' | trans }}
                <small>{{ 'rendered' | trans({'%date%':"now"|date("Y-m-d
                H:i:s")}) }}
                </small></h1>
            </div>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris
            consequat orci eget libero sollicitudin, non ultrices turpis mollis.
            Aliquam sit amet tempus elit. Ut viverra risus enim, ut venenatis
            justo accumsan nec. Praesent a dolor tellus. Maecenas non mauris leo.
            Pellentesque lobortis turpis at dapibus laoreet. Mauris rhoncus nulla
            et urna mollis, et lobortis magna ornare. Etiam et sapien consequat,
            egestas felis sit amet, dignissim enim.</p>
            <p>Quisque quis mollis justo, imperdiet fermentum velit. Aliquam
            nulla justo, consectetur et diam non, luctus commodo metus.
            Vestibulum fermentum efficitur nulla non luctus. Nunc lorem nunc,
            mollis id efficitur et, aliquet sit amet ante. Sed ipsum turpis,
```

```
            vehicula eu semper eu, malesuada eget leo. Vestibulum venenatis
            dui id pulvinar suscipit. Etiam nec massa pharetra justo pharetra
            dignissim quis non magna. Integer id convallis lectus. Nam non
            ullamcorper metus. Ut vestibulum ex ut massa posuere tincidunt.
            Vestibulum hendrerit neque id lorem rhoncus aliquam. Duis a
            facilisis metus, a faucibus nulla.</p>
        </div>
    </div>
    {{ render_esi(url('footer')) }}
</div>
</body>
</html>
```

This Twig template is loaded on line 46 of the preceding PHP code. It contains HTML that is styled by the Twitter Bootstrap library.

I'm also using the hinclude.js JavaScript library to perform so-called `hx:include` calls. These are placeholder tags similar to ESI tags. These tags aren't processed on the edge, but are instead processed by the browser. It's an alternative form of AJAX.

The header is rendered by calling `{{ render_hinclude(url(`*`header`*`)) }}`. This view helper renders a named route called `header` and displays it as an `<hx:include />` tag. This tag is processed by the browser and loads the `/header` route.

The next block that is loaded is the navigation. This is done by calling `{{ render_esi(url(`*`nav`*`)) }}`. This view helper loads a named route called `nav` and displays this block as an ESI tag. This tag is processed by Varnish and loads the `/nav` route.

The final block that is loaded is the footer, which is rendered by calling `{{ render_esi(url(`*`nav`*`)) }}` and this also loads a named route in similar fashion using ESI.

A final remark on the index template: you might have noticed `{{ `*`example`*` | trans }}` and `{{ `*`rendered`*` | trans({`*`%date%`*`:"now" | date("Y-m-d H:i:s")})}}`. These are translation keys that are translated by the trans filter. Depending on the language, the right message is parsed into these placeholders. We even inject the current date and time into the second placeholder.

The header template

The header was loaded as a separate HTTP fragment in the index template. This is the Twig template that is loaded when `/header` is called:

```
<div class="jumbotron">
    <div class="page-header">
        <h1>{{ 'welcome'|trans }}
        <small>{{ 'rendered' | trans({'%date%':"now"|date("Y-m-d H:i:s")}) }}
        </small></h1>
    </div>
</div>
```

As you can see, it is plain HTML enriched by some Twig placeholders and filters that provide translations.

The nav template

The navigation template is also a Twig template, but it just contains plain old HTML. Nothing special to report.

```
<nav class="navbar navbar-default navbar-fixed-side">
    <ul class="nav">
        <li><a href="#">Item 1</a></li>
        <li><a href="#">Item 2</a></li>
    </ul>
</nav>
```

The footer template

And finally, there's the footer template that displays a translated message and the current date, parsed into the message using the built-in Twig translation view helper.

```
<footer><small>{{ 'rendered' | trans({'%date%':"now"|date("Y-m-d H:i:s")}) }}
</small></footer>
```

Conclusion

The extensive code example is the perfect proof that it is possible to architect an application to be highly cacheable. If, for some reason, a part of the content is not cacheable, HTTP fragments can be used to load it on the edge or in the browser.

Live, eat, and breathe headers like `Cache-control`, `Vary`, `ETag`, `If-None-Match`, `Last-Modified`, `If-Modified-Since`, `Surrogate-Capability`, and `Surrogate-Control`. If you forgot about those, go all the way back to Chapter 3 and study them again!

If you're dealing with a legacy application where these headers are tough to implement, just write some VCL. It's OK.

CHAPTER 8

Logging, Measuring, and Debugging

Setting up Varnish, configuring parameters, and tuning your VCL is all fun and games, but how do you know how effective your setup is? You could look at the load statistics of your Varnish servers and your backends. If those are fine, you know your environment isn't on the verge of exploding. Unfortunately, this gives us little information on the hit rate of the cache—it would be nice to identify the *hot spots* in your application and whether those hot spots are cached appropriately. If it turns out that certain pages aren't cached when they should be, you would like to know why, right? Who knows, by optimizing your application or your VCL, the load could further drop and this could lead to a downsize of your infrastructure.

Luckily, Varnish offers ways to debug and measure the HTTP requests, the HTTP responses, and the cache. In this chapter, we'll cover the following tools:

- `varnishstat`
- `varnishlog`
- `varnishtop`

These binaries all use the shared memory log, which we talked about in "Shared log memory storage" on page 15.

> Varnish offers even more tools, but many are beyond the scope of this book. To learn more, you can have a look at the documentation for binaries like varnishhist and varnishncsa.

Varnishstat

The varnishstat binary displays statistics of a running Varnish instance. These are general statistics about connections, sessions, backends, storage, hit rate, and much more. This is a good dashboard for sysadmins: it shares little information about the application and individual requests, but paints a picture about the state of the Varnish instance. It's easy to see the number of objects in cache or space left on the storage device. You can even see if Varnish is dropping objects from the cache due to lack of space.

Example Output

Here's some example output of `varnishstat`.

```
Uptime mgt:   156+00:17:44                              Hitrate n:       10       66       66
Uptime child:  38+20:51:56                                 avg(n):   0.4300   0.3837   0.3837

    NAME                         CURRENT        CHANGE       AVERAGE        AVG_10       AVG_100
MAIN.uptime                  38+20:51:56
MAIN.sess_conn                   2360514          1.00             .          1.10          1.09
MAIN.client_req_400                  148          0.00             .          0.00          0.00
MAIN.client_req                 38137645          1.00         11.00         17.80         16.88
MAIN.cache_hit                  26460861          0.00          7.00         11.76         11.18
MAIN.cache_hitpass               3653130          1.00          1.00          2.70          2.66
MAIN.cache_miss                  4637227          0.00          1.00          2.04          1.81
MAIN.backend_busy                   6342          0.00             .          0.00          0.00
MAIN.backend_fail                     71          0.00             .          0.00          0.00
MAIN.backend_reuse              13202921          5.99          3.00          5.92          6.32
MAIN.backend_recycle            13264912         13.98          3.00          6.66          6.47
MAIN.backend_retry                 29828          0.00             .          0.00          0.00
MAIN.fetch_head                  1313793          1.00             .          0.40          0.40
MAIN.fetch_length                4365149          4.99          1.00          2.39          2.12
MAIN.fetch_chunked               6444020          5.99          1.00          3.46          3.06
MAIN.fetch_bad                         6          0.00             .          0.00          0.00
MAIN.fetch_none                   654444          3.00             .          1.01          0.88
MAIN.fetch_204                       559          0.00             .          0.00          0.00
MAIN.fetch_304                   1918036          0.00             .          0.26          0.78
MAIN.fetch_failed                     19          0.00             .          0.00          0.00
MAIN.pools                             2          0.00             .          2.00          2.00
MAIN.threads                         200          0.00             .        200.00        200.00
MAIN.threads_limited                   1          0.00             .          0.00          0.00
MAIN.threads_created                1292          0.00             .          0.00          0.00
MAIN.threads_destroyed              1092          0.00             .          0.00          0.00
```

Displaying Specific Metrics

The output you're getting doesn't include all the metrics: just execute `varnishstat -l` to see a full list.

Here's an example in which we output a limited set of metrics:

- All the information about the cache (hits, misses, hit-for-passes)

- The number of backend failures
- The number of bytes available in the cache
- The number of objects stored in cache
- The number of objects that were nuked from the cache due to lack of space

```
varnishstat -f MAIN.cache_* -f MAIN.backend_fail -f SMA.s0.c_bytes
-f MAIN.n_object -f MAIN.n_lru_nuked
```

And this could then be the output:

```
Uptime mgt:   156+00:32:10               Hitrate n:        9        9        9
Uptime child:  38+21:06:23                  avg(n):   0.4551   0.4551   0.4551

NAME                 CURRENT      CHANGE     AVERAGE        AVG_10       AVG_100
MAIN.cache_hit      26468969       53.94        7.00         13.74         13.74
MAIN.cache_hitpass   3654504       12.98        1.00          3.00          3.00
MAIN.cache_miss      4638817        4.99        1.00          3.75          3.75
MAIN.backend_fail         71        0.00           .          0.00          0.00
SMA.s0.c_bytes        164.34G      95.51K      51.30K        68.35K        68.35K
MAIN.n_object          48508      -11.99           .      48512.33      48512.33
```

Output Formatting

By default, `varnishstat` is visualized as a continuously updated list of metrics. You can use the `-1` flag to only get output once.

You can format the output as XML by adding the `-x` flag, and as JSON by adding the `-j` flag.

Varnishlog

Varnishlog is a binary tool that reads the shared memory logs and displays this information in real time. It represents the log as a list of tags and values. Have a look at the extensive list of VSL tags on the Varnish documentation site and learn how to interpret their value.

Example Output

This example is the unfiltered output of a `varnishlog` execution:

```
*   << Request  >> 65538
-   Begin          req 1 rxreq
-   Timestamp      Start: 1476957981.431713 0.000000 0.000000
-   Timestamp      Req: 1476957981.431713 0.000000 0.000000
-   ReqStart       10.10.10.1 60358
-   ReqMethod      GET
-   ReqURL         /
-   ReqProtocol    HTTP/1.1
```

```
-   ReqHeader      Host: example.com
-   ReqHeader      Connection: keep-alive
-   ReqHeader      Cache-Control: max-age=0
-   ReqHeader      Upgrade-Insecure-Requests: 1
-   ReqHeader      User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36
-   ReqHeader      Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
-   ReqHeader      Accept-Encoding: gzip, deflate, sdch
-   ReqHeader      Accept-Language: nl,en-US;q=0.8,en;q=0.6
-   ReqHeader      X-Forwarded-For: 10.10.10.1
-   VCL_call       RECV
-   VCL_return     hash
-   ReqUnset       Accept-Encoding: gzip, deflate, sdch
-   ReqHeader      Accept-Encoding: gzip
-   VCL_call       HASH
-   Hash           "/%00"
-   Hash           "example.com%00"
-   VCL_return     lookup
-   VCL_call       MISS
-   VCL_return     fetch
-   Link           bereq 65539 fetch
-   Timestamp      Fetch: 1476957982.211473 0.779760 0.779760
-   RespProtocol   HTTP/1.1
-   RespStatus     200
-   RespReason     OK
-   RespHeader     Date: Thu, 20 Oct 2016 10:06:21 GMT
-   RespHeader     Server: Apache/2.4.10 (Debian)
-   RespHeader     Vary: Accept-Encoding
-   RespHeader     Content-Encoding: gzip
-   RespHeader     Content-Length: 496
-   RespHeader     Content-Type: text/html; charset=UTF-8
-   RespHeader     X-Varnish: 65538
-   RespHeader     Age: 0
-   RespHeader     Via: 1.1 varnish-v4
-   VCL_call       DELIVER
-   VCL_return     deliver
-   Timestamp      Process: 1476957982.211516 0.779804 0.000043
-   RespHeader     Accept-Ranges: bytes
-   Debug          "RES_MODE 2"
-   RespHeader     Connection: keep-alive
-   Timestamp      Resp: 1476957982.211588 0.779875 0.000072
-   ReqAcct        424 0 424 289 496 785
-   End
**  << BeReq    >> 65539
--  Begin          bereq 65538 fetch
--  Timestamp      Start: 1476957981.431787 0.000000 0.000000
--  BereqMethod    GET
--  BereqURL       /
--  BereqProtocol  HTTP/1.1
--  BereqHeader    Host: example.com
--  BereqHeader    Upgrade-Insecure-Requests: 1
```

```
--  BereqHeader    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36
--  BereqHeader    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
--  BereqHeader    Accept-Language: nl,en-US;q=0.8,en;q=0.6
--  BereqHeader    X-Forwarded-For: 10.10.10.1
--  BereqHeader    Accept-Encoding: gzip
--  BereqHeader    X-Varnish: 65539
--  VCL_call       BACKEND_FETCH
--  VCL_return     fetch
--  BackendOpen    33 boot.default 127.0.0.1 8080 127.0.0.1 36222
--  Timestamp      Bereq: 1476957981.431879 0.000092 0.000092
--  Timestamp      Beresp: 1476957982.211306 0.779520 0.779427
--  BerespProtocol HTTP/1.1
--  BerespStatus   200
--  BerespReason   OK
--  BerespHeader   Date: Thu, 20 Oct 2016 10:06:21 GMT
--  BerespHeader   Server: Apache/2.4.10 (Debian)
--  BerespHeader   Vary: Accept-Encoding
--  BerespHeader   Content-Encoding: gzip
--  BerespHeader   Content-Length: 496
--  BerespHeader   Content-Type: text/html; charset=UTF-8
--  TTL            RFC 120 10 -1 1476957982 1476957982 1476957981 0 0
--  VCL_call       BACKEND_RESPONSE
--  VCL_return     deliver
--  Storage        malloc s0
--  ObjProtocol    HTTP/1.1
--  ObjStatus      200
--  ObjReason      OK
--  ObjHeader      Date: Thu, 20 Oct 2016 10:06:21 GMT
--  ObjHeader      Server: Apache/2.4.10 (Debian)
--  ObjHeader      Vary: Accept-Encoding
--  ObjHeader      Content-Encoding: gzip
--  ObjHeader      Content-Length: 496
--  ObjHeader      Content-Type: text/html; charset=UTF-8
--  Fetch_Body     3 length stream
--  Gzip           u F - 496 3207 80 80 3901
--  BackendReuse   33 boot.default
--  Timestamp      BerespBody: 1476957982.211449 0.779663 0.000143
--  Length         496
--  BereqAcct      406 0 406 196 496 692
--  End
```

You see tags like:

- `ReqURL`
- `ReqHeader`
- `VCL_call`
- `VCL_return`

- `Hash`
- `TTL`

All these tags and their respective values give you very detailed information about the state of a request. It shows you the request information, the different stages of the Varnish Finite State Machine, optional backend connections, the way an object is stored, the time-to-live, and much more.

This is an essential tool for debugging. I admit that it is extremely verbose, and on live systems, impossible to use without the proper filtering.

Filtering the Output

Luckily, `varnishlog` offers these filtering options:

`-g`
: Offers different kinds of output grouping

`-i`
: Allows you to only include certain VSL tags

`-I`
: Includes tags by regular expression

`-x`
: Allows you to exclude certain VSL tags

`-I`
: Excludes tags by regular expression

`-q`
: Allows you to execute a VSL query

In older versions of Varnish, debugging the shared memory log was difficult due to limitations in terms of filtering. In Varnish version 4, VSL queries were introduced. This feature made a huge improvement when it comes to debugging Varnish behavior.

Here's an example in which we want to see URLs that aren't served from cache, accompanied by the flow that is executed to get to that result:

```
varnishlog -i ReqUrl -i VCL_call -i VCL_return
-q "VCL_call eq 'MISS' or VCL_call eq 'PASS'"
```

So we're interested in log entries where the `VCL_call` tags equal either miss (cacheable, but not in cache) or pass (uncacheable). After filtering requests, we'll only display the log lines that contain the request URL (`ReqUrl`), the different stages of the Varnish

Finite State Machine (`VCL_call`), and the actions that were returned for every state (`VCL_return`).

```
-    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
```

This one request contained the following events:

1. Received by Varnish (`VCL_call RECV`)
2. Varnish decided it was cacheable and performed a cache lookup (`VCL_return hash`)
3. The lookup hash was composed (`VCL_call HASH`)
4. Based on the hash, a cache lookup was performed (`VCL_return lookup`)
5. The item was not found in cache and triggered a miss (`VCL_call MISS`)
6. A backend connection was made and the item was fetched (`VCL_return fetch`)
7. The fetched data is deliverable to the client (`VCL_call DELIVER`)
8. The response is effectively returned to the client (`VCL_return deliver`)

Varnishtop

Varnishtop is a binary that uses the same shared memory log as `varnishlog` and uses the same concept of tags and values. Even the binary options are nearly the same.

Then what makes it different? Well, the output: instead of outputting log lines as part of a request, it presents a continuously updated list of the most commonly occurring log entries. You can compare it to the top command in Linux.

Because `varnishtop` orders entries by number of occurences in the shared memory log, it doesn't make sense to run the command without applying the proper filters. So you'll use tag filtering and VSL queries to present meaningful output.

Here's an example in which we apply filters to `varnishtop` to retrieve a list of URLs with the most cache misses:

```
varnishtop -i ReqUrl -q "VCL_call eq 'MISS'"
```

This is the output it could return:

```
125.25 ReqURL         /page1
10.86 ReqURL         /page2
0.57 ReqURL         /
0.10 ReqURL         /page3
```

If you look at the output, you'll notice that `/page1` generates 125.25 misses per minute. It's worth having a look at that page to inspect its behavior. You can use `varnishlog` to check which `VCL_call` and `VCL_return` values it has while executing the flow.

> The default sampling rate is 60 seconds. If you want to change that value, you can use the `-p` option to modify the sampling rate of `varnishtop`.

Conclusion

If you run Varnish and don't use any of the tools mentioned in this chapter, you're vulnerable. You may be under the impression that all is good, but a sudden set of cache misses can come out of nowhere. Use `varnishtop` to keep tabs on your instances. You can even format the output in several ways. You can use these metrics in your monitoring agents and send alerts when values hit a critical level. You can easily audit the application behavior by issuing `varnishtop` commands, as they will allow you to see the most common executions of certain tags. If some of those values are out of the ordinary, you can dig deeper and use `varnishlog` to inspect very specific scenarios. If you really want to optimize your hit rate (as indicated in Chapter 7), you'll want to know the consequences of your VCL changes—these tools will help you to know for sure.

CHAPTER 9

What Does This Mean for Your Business?

By this point, you should have extracted enough information from this book to set Varnish up and to configure it appropriately. And if all goes well, your application will have been tuned a bit, as well.

But why should you do this? From a technical perspective, it's an easy decision: to make your platform better, faster, and stronger. But from a business perspective, this can be a tough sell. If you're a developer or sysadmin and you're looking to convince your manager to use Varnish, this chapter will give you the ammo you need. If you're a technical decision-maker interested in using Varnish, this chapter is your source of inspiration.

This chapter offers some success stories and a bit of practical advice on how Varnish can fit into your stack—and into your organization.

To CDN or Not to CDN

A content delivery network (CDN) is nothing more than a set of reverse caching proxies that are hosted in various locations. A CDN is the perfect example of caching on the edge.

Companies often play the CDN card because it's offered as a service and they get support. And in many cases, they go for a CDN purely for caching, not for geo distribution. The downside is the price, but because they're in crisis mode, money is not an issue.

At Combell and at Sentia, the brands I work for, we managed to convince some of our clients to stop paying for a CDN and just use Varnish instead. Because of the power of VCL, we were able to achieve a much better hit rate and we had more insight because of tools like `varnishlog`. And they only paid a fraction of the money to host Varnish that they would have for a CDN.

I'm not saying CDNs are bad. Heck no! Varnish can be a good alternative for expensive CDNs, but it all depends on the context of your project. Please consider it, though. Some of our bigger clients need a CDN, either because they have a global presence that requires multiple points-of-presence, or because the sheer number of connections was saturating some network devices. And some CDNs are great at averting DDoS attacks—unless you have huge networking resources, those are tough to fend off.

> Although I'm positioning Varnish as a competitor of CDNs, many CDNs actually use Varnish for their reverse caching proxies. Varnish Software even has a product called Varnish Extend that allows you to set up your own private or hybrid CDN.
>
> In fact, this is an easy way to extend the reach of your current CDN: Varnish Extend allows you to add servers to your current CDN in regions where there is no point of presence, basically creating a hybrid setup. Because it's just software, you can install Varnish Extend on servers of your preference in a location of your preference.

VCL Is Cheaper

When Varnish is implemented, it's often in times of crisis: the servers are under heavy load and the website is slow or unavailable. There isn't much time to waste. Refactoring code or optimizing database queries is not an option.

Desperate times make even the most conservative manager take a second look at technology that was previously not desired. Where open source software was often stigmatized as being amateuristic, it now becomes a genuine option. Any lifeline will be accepted at this point. And although I've been preaching application-level HTTP best practices throughout the book, I'm happy to admit that VCL is cheaper in these cases. Even a couple of lines of quick and dirty VCL code can be a lifesaver.

It's under these circumstances that decision-makers start to appreciate Varnish. Point out to your manager the simple syntax of VCL as illustrated in Chapter 4. Show your manager some common VCL scenarios as illustrated in Chapter 7. Compare the limited cost and effort to the alternative and you'll see that a little bit of VCL goes a long way and is tough to beat.

If you make a solid case, you can bet your life that your manager's next project will have Varnish in the web stack. Organizations that prior to a crisis were unaware of headers like `Cache-control` will have a pretty solid idea of how they can leverage HTTP in future projects.

Varnish as a Building Block

We went from not having Varnish to having Varnish when chaos arises. I call that an improvement. But the real lesson we need to learn here is: why didn't we have a reverse caching proxy in the first place? As mentioned in "Caching Is Not a Trick" on page 4: don't recompute if the data hasn't changed. Why not treat caching as a strategy?

Many open source projects, such as Wordpress, Drupal, and Magento, offer Varnish support. These Varnish plugins know how to invalidate the cache, come with a VCL file, and in some cases offer support for block caching.

Reverse caching proxies are becoming a commodity. Agreed, if you run dedicated Varnish servers, it drives up cost. But the money you save by using Varnish outweighs these expenses. And if you want to be cheap, you can just install Varnish on your local web server—there's nothing wrong with that.

I work in the hosting industry and notice that sales people and solution architects are very quick on the trigger when it comes to suggesting Varnish. In most cases, it's a no-brainer. Sometimes we have a look at the application and either advise some refactoring or just write some VCL to get a decent hit rate.

There are development frameworks like Symfony that come with a built-in reverse caching proxy that obeys the same basic rules as Varnish. It's an interesting point of view: Varnish is just an implementation. It doesn't always matter what kind of technology you use, as long as it respects conventions and does the job.

On local development machines, Varnish isn't always available. Having a built-in reverse cache proxy is definitely a convenience. And when it behaves the same way as Varnish does, using Varnish on test/staging/production isn't a high-risk operation.

Pushing companies towards a "caching state of mind" will result in faster response times. See "Why Does Web Performance Matter?" on page 1.

The Original Customer Case

Varnish wasn't originally a pure open source project; it was a custom piece of software built for VG.no, the biggest news website in Norway. They got Poul-Henning Kamp on board to build the thing and Linpro to port it to Linux, provide the tooling, and build the website. Varnish Software eventually spun off from Linpro and is now the commercial entity of the project.

The reverse caching proxy that was built for *VG.no* proved to be a real success and the code was stable and clean enough to be open sourced.

Without *VG.no*, there would be no Varnish and there wouldn't be any other customer cases.

Varnish Plus

There are some mission-critical aspects of online businesses that Varnish doesn't support out of the box. These require lots of custom VCL code, a bunch of VMODs, and some infrastructure trickery.

Varnish Plus is a commercial product by Varnish Software that adds business-oriented and mission-critical features on top of Varnish.

These are some of the features that Varnish Plus offers as a service:

- High availability
- Massive storage engine
- Enhanced cache invalidation
- Varnish Plus support
- Built-in SSL/TLS support
- Mobile device detection
- An easy-to-use administration console
- Paywall support
- Custom statistics
- Edgestash
- Parallel ESI

You'll agree that these are features that are tailored around the needs of businesses. You can build all these features yourself using VCL and VMODs, but it would take you a lot more time. Varnish Plus offers these advanced features on top of Varnish, with support and an intuitive management interface.

Companies Using Varnish Today

There are thousands of companies using Varnish: small, big, and anything in between. In this section, I'll highlight a couple of interesting customer cases where it's not just about the company, but about what they do with Varnish.

NU.nl: Investing Early Pays Off

By investing in caching and Varnish early on, you'll save money in the long run. By having tight control over your caches and the know-how to invalidate when required, you'll avoid spending money on extra servers for horizontal scalability purposes.

I love the NU.nl case study by Ibuildings that explains how an agressive caching strategy allowed the company to cope with massive traffic to the Nu.nl news site when an airliner crashed at Schiphol Airport in 2009. Pictures from bystanders were shared on the website and traditional media outlets couldn't keep up with the pace.

The NU.nl website was the global center of attention that day with more than 20 million page views. And the site did not collapse because Varnish was able to serve all content from cache. It only needed two Varnish servers to handle that kind of load.

Although this is an old case that dates back to 2009, I still like to tell the story. The Ibuildings folks made a presentation about it and named it Surviving a plane crash. Talk about saving money on infrastructure cost!

SFR: Build Your Own CDN

SFR is a French telecommunications company that provides voice, video, data, and internet telecommunications and professional services to consumers and businesses. SFR has 21 million customers and provides 5 million households with high-speed internet access.

SFR's main website is one of the top-10 most-visited sites in France. The costs of handling traffic were high enough that SFR decided to look into cost-reduction initiatives and ways to bring costs and efforts in-house.

SFR decided to build its own CDN using Varnish. The company started out with regular Varnish, but quickly got in touch with Varnish Software and ended up using Varnish Plus. SFR has so many objects to cache that regular Varnish storage options did not meet its requirements. The Massive Storage Engine ended up solving that problem.

SFR also added some VMODs that allowed bandwidth throttling and session identification for users who consumed video that was served from the CDN.

Varnish at Wikipedia

Even Wikipedia uses Varnish. Emanuele Rocca, one of Wikipedia's operations engineers, gave a presentation at Varnishcon 2016 about how Wikipedia uses Varnish. The slides and video footage are available online.

Wikipedia chose Varnish because Varnish is open source and has a proven track record. The Wikimedia foundation itself is a very strong advocate of open source software and has deep roots in the free culture and free software movements.

Basically, Wikipedia has a multitiered Varnish stack to handle on average 100,000 incoming requests per second. It uses Varnish for its own multiregion CDN, but also for page caching. Because of Wikipedia's values, it wants full autonomy and doesn't want to depend on external companies. It also has some technical requirements that require full control over the caching technology. An as-a-service model doesn't really work for Wikipedia.

Varnish is the ideal tool for this usage for the following reasons:

- Wikipedia has full control over caching and purging policies
- VCL allows it to write many custom optimizations
- Varnish allows it to have custom analytics

Wikipedia's multitier Varnish setup consists of two varnishd instances that run on separate ports:

- A first tier that stores objects in memory
- A second tier that stores objects on disk

Incoming requests are routed to the "best" data center based on GeoDNS. Load balancers send traffic to a caching node based on the client IP. Because TLS termination happens on the caching nodes and because of TLS session persistence, it makes sense to send requests from the same client IP to the same node.

The first Varnish tier serves popular content from memory. If an object is not in cache, the request is sent to the second tier that stores its cache on disk. This is still a lot faster than serving it directly from the web server. Requests are routed to the second Varnish tier based on the URL. This allows you to shard data across multiple servers without having duplicate objects.

Because of object persistence in the second tier, Wikipedia can easily cope with restarts without the risk of losing cached data.

Combell: Varnish on Shared Hosting

At Combell, we even offer Varnish on our shared hosting environment. We do this for entirely different reasons than what the previous cases would suggest.

The NU.nl, SFR, and Wikipedia cases had an underlying theme of massive traffic and how Varnish allows these companies to cope with that. At Combell, we offer Varnish on our shared hosting not for scalability purposes, but to get the raw performance.

Let's face it: shared hosting is not exactly built to handle lots of traffic. But a lot of small websites can get really slow because of poor technical design or poor execution in terms of code. By adding Varnish to the stack, these slow requests can be sped up—increased scalability is merely a bonus.

But due to caching and the design of our platform, we run websites on this shared environment that can easily handle 50,000 visitors per day. When some of these websites have a good hit rate, Varnish allows them to have many more visitors without the slightest delay.

Conclusion

Varnish is a serious project—a serious piece of technology for serious organizations. Because of its simplicity and flexibility, it's even a good fit for smaller companies and organizations that have performance rather than scalability issues. Varnish has an impressive track record and is used by more than 2.5 million websites, some of which are among the most popular in the world.

If you're looking to integrate Varnish into your projects but you still encounter some resistance from colleagues or management, show them these customer cases. If it's good enough for Wikipedia, it's probably good enough for your website.

And if all of this is still way too complicated and technical for the person in charge, let them watch this video explaining Varnish cache, then talk about success stories, and finally create a proof of concept with a bit of VCL.

If your management wants more formal guarantees and extra services, point them in the direction of Varnish Software.

I like Varnish. The industry likes Varnish. Hopefully, by now, you like it, too!

CHAPTER 10

Taking It to the Next Level

By now, we've established a level of understanding:

- You know what Varnish is and where it fits in
- You know how to install and configure Varnish
- You understand some of the HTTP best practices that Varnish respects
- You can interprete VCL snippets and write some of your own
- You know how to invalidate (parts of) the cache
- You're familiar with the load-balancing capabilities of Varnish
- You're able to leverage some of the Varnish logging and monitoring tools
- You know how to adapt your application or your VCL to real-world scenarios

So what's next? This chapter will point you in the right direction if you want to take it to the next level. I'll address some specific topics that are beyond the scope of this book and refer to useful resources.

We'll talk about RESTful services, VMODs, the future of the Varnish project, and ways to get help if you have questions.

What About RESTful Services?

Throughout this book, I've been focusing on websites and web applications. But in this day and age, service-oriented architectures have become normal. The idea that software isn't only consumed by humans but also by other systems has become mainstream. The fact that your application has a feed or an API doesn't seem that exotic anymore. Nowadays, the application usually *is* a REST API and the GUI is just a piece of frontend code that consumes it.

That being said, RESTful services are very easy to cache with Varnish. As a matter of fact, Roy Fielding, who introduced REST in his PhD dissertation, defined REST as "an architectural style to describe the design and development of the architecture for the modern web."

In his dissertation, he applies REST to the HTTP protocol and describes his experiences. He also refers to several data elements that can be used to identify and cache resources.

So in essence, if you write true RESTful services, you are already applying the best practices and you don't need a lot of VCL changes. The advice from Chapter 3 surely applies here, as well!

Patch Support

You will need to add explicit support for the HTTP PATCH method. Varnish 4.1 only supports the HTTP methods that were defined in RFC 2616 and PATCH is not part of that specification. PATCH support was added in RFC 2068, you'll need to provide the following piece of VCL to make it work:

```
if (req.method != "GET" &&
    req.method != "HEAD" &&
    req.method != "PUT" &&
    req.method != "POST" &&
    req.method != "TRACE" &&
    req.method != "OPTIONS" &&
    req.method != "PATCH" &&
    req.method != "DELETE") {
    return (pipe);
}
```

Authentication

If access to your RESTful API requires authentication, you can either use the `authorization` header or use a VMOD for alternative forms of authentication. We'll talk about VMODs in "Extending Varnish's Behavior with VMODs" on page 133.

But if you want simple basic authentication on your API without having to connect to the backend, you could use `vmod_basicauth`.

Invalidation

Letting your RESTful API return stale content is just as risky as it would be on websites and web applications. Therefore, it is equally important to have a correct invalidation strategy.

There's obvious low-hanging fruit like invalidating individual resources that are called using `POST`, `DELETE`, `PATCH`, and `PUT`.

Let's say you perform a PUT as in this example:

```
PUT /user/1
```

You're basically performing a full update on the `/users/1` resource. So if you're trying to retrieve that resource using GET as in the following, you need to make sure it is invalidated:

```
GET /user/1
```

If you remember the built-in VCL, you'll know that GET calls are cached and PUT calls are not. In your PUT logic, you can purge or ban that resource.

But it gets more complicated: there are aggregate resources that could also contain information about the updated resource. Take, for example, the following GET call:

```
GET /user
```

This `/user` resource also needs to be invalidated as it contains a collection of users that includes user 1.

In summary, as RESTful APIs are data-driven, it is usually important that the data is consistent. Make sure you have a decent cache-invalidation strategy. If eventual consistency is acceptable, you can rely on the expiration of your resources without having to worry about explicit invalidation.

Extending Varnish's Behavior with VMODs

As you will remember from Chapter 4, the VCL syntax is quite extensive and offers many ways to influence the behavior of Varnish. But depending on your use case, you might encounter some limitations—maybe VCL doesn't support a feature you need.

In previous versions of Varnish, the use of inline C-code was promoted. Because Varnish compiles VCL into C code, it was quite easy to process additional C code. But as you can imagine, the slightest of errors would crash Varnish entirely.

> In Varnish 4.1, inline C is still supported, but not advised. If you want to use custom C code, you need to enable `vcc_allow_inline_c` as a startup option.

So what's the alternative? Varnish now promotes the use of Varnish modules, also known as VMODs. A VMOD is also written in C, but instead of directly extending the behavior of Varnish, a VMOD is offered as a shared library that can be called from your VCL code. You could say that VMODs are modules that enrich the VCL syntax with additional features.

Finding and Installing VMODs

As mentioned before, there is an official VMOD directory on the Varnish website. You can also write your own VMODs if you need to. The Varnish community offers some basic VMODs that are installed as one big collection. They're up on GitHub if you're interested.

Enabling VMODs

It's not because you compiled and installed the VMODs in the right directory that they are immediately enabled. You still have to import them into your VCL file.

Here's how you import the standard VMOD:

```
import std;
```

Depending on the kind of VMOD, you can then start using its functions within the VCL subroutines. If your VMOD requires further initialzation, you can do this in `vcl_init`.

Remember this example from Chapter 6?

```
vcl 4.0;

import directors;

backend web001 {
  .host = "web001.example.com";
  .probe = healthcheck;
}

backend web002 {
  .host = "web002.example.com";
  .probe = healthcheck;
}

sub vcl_init {
    new loadbalancing = directors.round_robin();
    loadbalancing.add_backend(web001);
    loadbalancing.add_backend(web002);
}

sub vcl_recv {
    set req.backend_hint = loadbalancing.backend();
}
```

In that example, we needed to initialize a director by setting its distribution algorithm and adding backends. We did this in `vcl_init`.

VMODs That Are Shipped with Varnish

If you installed Varnish, you already have two VMODs at your disposal:

- vmod_std that offers a set of standard functions
- vmod_directors that offers load-balancing capabilities

Everything beyond that needs to be installed separately.

Need Help?

Read the book and feel like you have all the tools you need to setup a Varnish stack with a killer hit rate? Good for you! This book doesn't have the ambition or arrogance to be the *Varnish bible*. There are still things to be learned.

But if you're stuck and need help, here are several available resources:

- varnish-misc mailing list is open for questions
- There is a #varnish IRC channel on irc.linpro.no. People there tend to be friendly and helpful
- There is a Varnish tag on StackOverflow where you can ask questions
- VCL-related questions can be asked on the varnish-vcl tag on StackOverflow
- If you're a #dta kind of person and you just want to throw it on Twitter, you can also direct your questions to @varnishcache

If you need support, training, or professional services, Varnish Software is the company you need. Varnish Software is the main sponsor of the Varnish project and employs a lot of Varnish core contributors. They're the one-stop shop when it comes to commercial products and services on top of Varnish.

The Future of the Varnish Project

By the time you're reading this, Varnish version 5 will be out and about, but that does not mean this book is outdated. The adoption rate for the broad public is usually quite slow. There are still tons of Varnish setups out there that use version 3.

Varnish 5 features a couple of changes, but nothing that is too worrying. The VCL syntax remains unchanged.

The Varnish team has now switched from a feature-based release schedule to a time-based release schedule. This means you can expect a new version of Varnish every six months. The main new feature in Varnish 5 is experimental support for HTTP/2. It's not yet production-ready, but it can handle HTTP/2 traffic if you enable the http2 feature flag.

There are also some other features and improvements. I wrote a blog post about it—check it out!

With the two annual releases we can expect, it will be easier for users to see the direction Varnish is heading in. I expect HTTP/2 to remain the main talking point. I expect a lot of improvements by the next release when it comes to HTTP/2 support.

The future of Varnish is bright. Stay tuned, and keep an eye out for new releases!

Index

Symbols

A

B

C

I

L

M

N

O

P

Q

R

S

T

U

V

W

About the Author

Thijs Feryn is a technical evangelist at a Belgian web hosting company called Combell. His goal is to bring technology to the people and people to technology. He focuses on bridging the gap between code and infrastructure. Thijs is also involved in many open source communities. He speaks, listens, writes, codes, teaches, organizes, and is, above all, very excited about this book.

Colophon

The animal on the cover of *Getting Started with Varnish Cache* is *Montagu's harrier* (*Circus pygargus*); its name pays homage to British naturalist George Montagu.

The harrier's plumage differs based on sex: adult males are covered in a pale grey plumage with black wingtips, and adult females have pale yellow underparts with longitudinal stripes and uniformly dark brown upper parts. As a juvenile, all harriers resemble adult females but with uniformly red brown under parts.

The Montagu's harrier is known for its graceful flight and powerful wingbeats. This harrier is small compared to most raptors; its wingspan ranges 38–45 inches and it weighs between 265 grams (males) and 345 grams (females). The female of the species is larger in order to produce eggs.

This species is found in temperate climates as well as in the Mediterranean and boreal zones. It tends to nest in lowlands, mostly broad river valleys, plains, and areas bordering lakes or the sea. When desperate for habitat, the Montagu's harrier will nest in agricultural farmlands, making it vulnerable to early harvesting. This harrier eats mostly small rodents, birds, bird eggs, reptiles, and large insects, taking whatever prey is available in the area of its nest. It catches prey while flying at low heights and low constant speeds along fixed routes. During the breeding season, males provision food for the females and later for the young. Males tend to hunt over a large area—within 7.5 miles from the next. Females often hunt closer to the nest, within 0.62 miles away, and only after the young have hatched.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *British Birds*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Getting Started with Varnish Cache

How long does it take for your website to load? Web performance is just as critical for small and medium-sized websites as it is for massive websites that receive tons of hits. Before you pour money and time into rewriting your code or replacing your infrastructure, first consider a reverse-caching proxy server like Varnish. With this practical book, you'll learn how Varnish can give your website or API an immediate performance boost.

Varnish mimicks the behavior of your webserver, caches its output in memory, and serves the result directly to clients without having to access your webserver. If you're a web developer familiar with HTTP, this book helps you master Varnish basics, so you can get up and running in no time. You'll learn how to use the Varnish Configuration Language and HTTP best practices to achieve faster performance and a higher hit rate.

- Understand how Varnish helps you gain optimum web performance
- Use HTTP to improve the cache-ability of your websites, web applications, and APIs
- Properly invalidate your cache when the origin data changes
- Optimize access to your backend servers
- Avoid common mistakes when using Varnish in the wild
- Use logging and debugging tools to examine the behavior of Varnish
- Craft a single client application that can consume multiple services

"Thijs Feryn eases the rather steep learning curve associated with Varnish Cache. The software is written entirely differently than most other software out there. This book does a great job of communicating how these differences make Varnish Cache a uniquely flexible tool for enhancing web performance."

—Per Buer
Founder and CTO at Varnish Software

Thijs Feryn is a technical evangelist at Combell, a Belgian web-hosting company where he focuses on bridging the gap between code and infrastructure. Thijs is involved in many open source communities and, as an established international conference speaker, he's delivered more than 160 presentations in 15 countries. Visit *https://feryn.eu* for more information about Thijs and his work.

WEB PROGRAMMING

9 781491 977187

Twitter: @oreillymedia
facebook.com/oreilly