



INSTANT
Short | Fast | Focused

Varnish Cache How-to

Hands-on recipes to improve your website's load speed
and overall user experience with Varnish Cache

Roberto Moutinho

[PACKT]
PUBLISHING

www.it-ebooks.info

Instant Varnish Cache How-to

Hands-on recipes to improve your website's load speed
and overall user experience with Varnish Cache

Roberto Moutinho



BIRMINGHAM - MUMBAI

Instant Varnish Cache How-to

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2013

Production Reference: 1210113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-040-3

www.packtpub.com

Credits

Author

Roberto Moutinho

Project Coordinator

Amigya Khurana

Reviewer

Wilson França

Renata Moutinho

Proofreaders

Maria Gould

Lynda Sliwoski

Acquisition Editor

Wilson D'souza

Graphics

Aditi Gajjar

Commissioning Editor

Priyanka Shah

Production Coordinator

Prachali Bhiwandkar

Technical Editors

Varun Pius Rodrigues

Sadhana Verma

Cover Work

Prachali Bhiwandkar

Cover Image

Sheetal Aute

About the Author

Roberto Moutinho is a system analyst, DevOps evangelist, and a open source and open standards advocate currently residing in Rio de Janeiro, Brazil. Passionate about the Web, he's currently a web developer working in one of the largest classified platform in Latin America, optimizing overall performance of in-house and third-party services. When off duty, you'll often find him taking apart old electronics to salvage components for weird and often useless DIY projects.

The success of this project depends largely on the encouragement and guidelines of my sister Renata Moutinho. I can't thank her enough for her tremendous support and help. I'd also like to thank my family, especially my father Jose Jorge J. Moutinho and my mother Sonia M. Monteiro Moutinho, for their constant support and help, all my friends and coworkers for their ideas and feedback during the writing process of this book, and the guidance and support received from all members of Packt Publishing.

About the Reviewer

Wilson França is a software engineer with over a decade of experience in developing systems. He has worked in several different areas from billing systems for the first national mobile carrier in Brazil to real-time ticket systems for the largest beverage producer in the world, and lately he has been developing Rest APIs for the largest comparison shopping portal in Latin America. In his spare time, he is a training instructor for the Federal Technology Center (CEFET-RJ) and an Arduino enthusiast.

First, I would like to thank to Roberto Moutinho who gave me the opportunity to collaborate in this project. I would like to express my special gratitude to Packt Publishing for giving me such attention and time. My thanks and appreciation also go to my family and people who have helped me out with their abilities.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Instant Varnish Cache How-to	5
Installing Varnish Cache (Must know)	5
Varnish Cache server daemon options (Must know)	7
Connecting to backend servers (Should know)	9
Load balance requests (Should know)	14
The Varnish Configuration Language (Should know)	16
Handling HTTP request vcl_recv (Should know)	20
Handling HTTP request vcl_hash (Should know)	23
Handling HTTP request vcl_pipe and vcl_pass (Should know)	25
Handling HTTP response vcl_fetch (Should know)	26
Handling HTTP response vcl_deliver (Should know)	29
Handling HTTP response vcl_error (Should know)	31
Caching static content (Should know)	33
Cookies, sessions, and authorization (Become an expert)	34
HTTP cache headers (Should know)	36
Invalidating cached content (Should know)	38
Compressing the response (Become an expert)	40
Monitoring the hit ratio (Become an expert)	41

Preface

Welcome to *Instant Varnish Cache How-to*.

Varnish Cache is an HTTP accelerator that stands between your customers and your application servers, acting as a front-end caching solution to your website. Varnish Cache is not a complete solution and will not serve your application content on its own; instead, it will proxy customers' requests to your web server and store their responses for later usage, avoiding an unnecessary (duplicated) workload.

Putting your website behind a Varnish Cache is quick and easy. In this book, you will learn how to make the most out of your infrastructure with minimal effort.

This book is the result of a year of collected information about scalability issues and caching in general, which led to the deployment of Varnish Cache on one of the major e-commerce/marketplace in Latin America.

What this book covers

Installing Varnish Cache (Must know), introduces you to how to install Varnish Cache using its own official repository and make sure everything is set before we start configuring our Varnish daemon.

Varnish Cache server daemon options (Must know), explains all the daemon options which make your Varnish Cache server act as expected, and adjust memory and CPU usage.

Connecting to backend servers (Should know), defines from which backend servers Varnish will fetch the data and also create mechanisms to make sure that Varnish does not proxy a request to any offline backend.

Load balance requests (Should know), explores all the possibilities of load balancing requests through application servers and creating a dedicated pool of servers to critical parts of your application system.

The Varnish Configuration Language (Should know), will get you started on writing VCL code to manage all your caching policies and manipulate requests and responses.

Handling HTTP request vcl_recv (Should know), explains how to choose backend servers according to what was requested by the client, block restricted content, and avoid cache lookup when it should not be performed.

Handling HTTP request vcl_hash (Should know), explains how Varnish generates the hash used to store the objects in memory and how to customize it to save memory space.

Handling HTTP request vcl_pipe and vcl_pass (Should know), introduces how to handle requests that should not be cached, making the data flow directly to the client.

Handling HTTP response vcl_fetch (Should know), provides the opportunity to rewrite portions of the originated response to suit your needs and stitch ESI parts.

Handling HTTP response vcl_deliver (Should know), explains how to clean up extra headers for server obfuscation and add debug headers to responses.

Handling HTTP response vcl_error (Should know), introduces how to deliver maintenance pages during scheduled or unexpected downtime or redirect users.

Caching static content (Should know), explains how to identify portions of your website that can take advantage of a cache server and how to cache them.

Cookies, sessions, and authorization (Become an expert), explains how to identify unique users and parts of the website that should never be cached.

HTTP cache headers (Should know), discloses what headers are important to a cache server and how to deal with their absence.

Invalidating cached content (Should know), explains how to invalidate content that should no longer be delivered and how to avoid it.

Compressing the response (Become an expert), explains how to save bandwidth and improve the overall speed by compressing data.

Monitoring hit ratio (Become an expert), discloses how well your cache is performing and how Varnish is connecting to your backend servers.

What you need for this book

To use this book effectively, you need the following software and operating system:

- ▶ Linux CentOS
- ▶ Varnish Cache
- ▶ Oracle VirtualBox

Who this book is for

This book targets system administrators and web developers with previous knowledge of the HTTP protocol who need to scale websites without spending money on large and costly infrastructure.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We will install Varnish Cache on a Linux CentOS box using the `varnish-cache.org` repository."


A block of code is set as follows:


```
set obj.http.Content-Type = "text/html; charset=utf-8";
set obj.http.Retry-After = "5";
synthetic {"
```

Any command-line input or output is written as follows:

```
# sudo yum install varnish
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In case of an unexpected failure, you can redirect costumers to an **Oops! We're sorry** friendly page."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Varnish Cache How-to

Welcome to *Instant Varnish Cache How-to*. In this book, we will cover the basics of setting up a Varnish Cache server instance in front of your website, how to identify cacheable portions of it, and how to get the best performance from your cache mechanism and policies.

Varnish Cache is a caching reverse proxy—often referred to as an HTTP accelerator, which sits between your application server and the client's requests. Its main goal is to avoid unnecessary duplicated work when the generated response is known to be the same, and with its flexible framework, it allows you to manipulate requests and also stitches together the **Edge Side Includes (ESI)** parts.

Most of the sample codes found in this book were written for a specific use-case and should be applied with caution. Hopefully, these examples will inspire you to write your own solutions for your scenario, and deploy a fast and reliable Varnish Cache inside your infrastructure.

This book targets system administrators and web developers with previous knowledge of the HTTP protocol. I've made a few assumptions throughout the book regarding the HTTP protocol and if you ever find yourself in need of an extended explanation, you can always refer to the HTTP Version 1.1 documentation at <http://www.w3.org/Protocols/rfc2616/rfc2616.html> or the Varnish Cache 3.0 documentation at <https://www.varnish-cache.org/docs/3.0/>.

Installing Varnish Cache (Must know)

The Varnish Cache binaries are very likely to be provided by your Linux distribution package repository, but the version you get from that repository might not be as up-to-date as it should be.

The Varnish Cache official repository provides versions for Red Hat- or Debian-based distributions, FreeBSD, and there's also the possibility to install it manually from a tarball file.

In the following recipe, we will install Varnish Cache on a Linux CentOS box using the `varnish-cache.org` repository and the **yum** package manager.

Getting ready

The following example will use both Varnish Cache 3.0.3 and 64-bit Linux CentOS 6.

It's recommended to try it out first on a virtual machine, since your caching policy will need to be adjusted and debugged before you go live.

If you don't have virtualization software yet, I recommend the Oracle VirtualBox at <https://www.virtualbox.org/> for its easy-to-use interface.

For other Linux distributions, you can find the correct `varnish-cache.org` repository at <https://www.varnish-cache.org/releases/>.

How to do it...

1. Add the `varnish-cache.org` repository to your CentOS box by typing the following command:

```
# sudo rpm --nosignature -i http://repo.varnish-cache.org/redhat/varnish-3.0/el5/noarch/varnish-release-3.0-1.noarch.rpm
```
2. Install Varnish Cache by typing the following command:

```
# sudo yum install varnish
```
3. Start the Varnish Cache daemon by typing the following command:

```
# sudo service varnish start
```

How it works...

Adding the `varnish-cache.org` repository allows us to keep our varnish server up-to-date and running with stable versions only.

If there's no `varnish-cache.org` repository for your Linux distribution, or in case you decide to compile it by hand, you'll need to download a tarball file from <http://repo.varnish-cache.org/source/> and resolve the dependencies before you use the GNU `make` and `make install` commands.

There's more...

Always check if the startup script service was correctly added to the runlevels list by typing the following command:

```
# sudo chkconfig --list varnish
```

You should see runlevels 2, 3, 4, and 5 marked as `on`, as follows:

```
varnish0:off 1:off 2:on 3:on 4:on 5:on 6:off
```

In case the service is turned `off` in any of these runlevels, turn them `on` by using the following command:

```
# sudo chkconfig varnish on
```

This will ensure that in case of a power outage or accidental server restart, our Varnish Cache instance will be up and running as soon as possible.

Startup scripts (also known as initialization script), are scripts that are typically run at system boot time. Most of them start services and set initial system parameters. For more information, visit https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/s2-boot-init-shutdown-init.html.

The runlevels list determines which programs are executed at system startup. More information about how runlevels works can be found at https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/s1-boot-init-shutdown-sysv.html.

Varnish Cache server daemon options (Must know)

Now that we have installed the Varnish Cache service and it's running, we need to take a moment to make sure that our startup parameters are correct.

The default post-installation storage method declared in the startup script is `file`, and we will change that to a memory-based storage in order to maximize our performance boost.



Bear in mind that our entire data is supposed to fit into memory. In case it doesn't, stick with the default storage type (`file`).

Some parameters such as the number of open files, the location of the configuration file, and others do not require adjusting to server specifications and should work as provided.

The daemon options go from setting the amount of space for storage and the type of storage used up, to thread pooling and hashing algorithms.

Getting ready

The `varnish` file at `/etc/sysconfig` is where all daemon options are conveniently located and also the resource for the startup script. You can also start the daemon manually and set the configuration parameters by passing arguments to it, but there's no need to do it since everything is already packed and ready. Every change to the `varnish` file at `/etc/sysconfig` is only valid after a full restart of the Varnish Cache service.

To restart the Varnish Cache, use the following provided script:

```
# sudo service varnish restart
```



When you perform a full restart, the cache is completely wiped. Be careful.

How to do it...

1. Open the `varnish` file from `/etc/sysconfig` using your favorite text editor (I'm using `vim`) by typing the following command:

```
# sudo vim /etc/sysconfig/varnish
```

Take your time and read all the comments so that you can understand every bit of this file, since it's loaded by the startup script (`/etc/init.d/varnish`).

2. Find the `VARNISH_STORAGE` parameter and change the default value to `malloc`, `${VARNISH_STORAGE_SIZE}`.

```
VARNISH_STORAGE="malloc,${VARNISH_STORAGE_SIZE}"
```

3. Set the `VARNISH_STORAGE_SIZE` parameter to about 85 percent of your available ram.

On a 16-GB RAM system, we can allocate 14-GB for storage and leave the remaining 2-GB for OS usage by using the following command:

```
VARNISH_STORAGE_SIZE=14G
```

How it works...

Both methods of storage—file and malloc—make use of file and memory resources but in a slightly different manner.

While the file storage type will allocate the entire cache size on a file, and tell the OS to map that file to memory (if possible) in order to gain speed, the malloc will request the storage size to the OS, and let it decide about how to divide and swap to file, what it can't fit into memory.



Don't get fooled by the name

The file storage does not keep data in file when you restart your Varnish Cache server.

There's more...

A new and still experimental storage type called persistent will work in a similar fashion for the file storage, but not every object will persist, since it can't handle situations where there's no more space left on the disk. The main benefit of this new storage type would be having a warmed up cache when recovering from an outage, since the objects are still available on disk.

The warm-up phase can also be performed with a tool called `varnishreplay`, but it requires much more time, since you will need an access logfile to replay it to Varnish Cache.

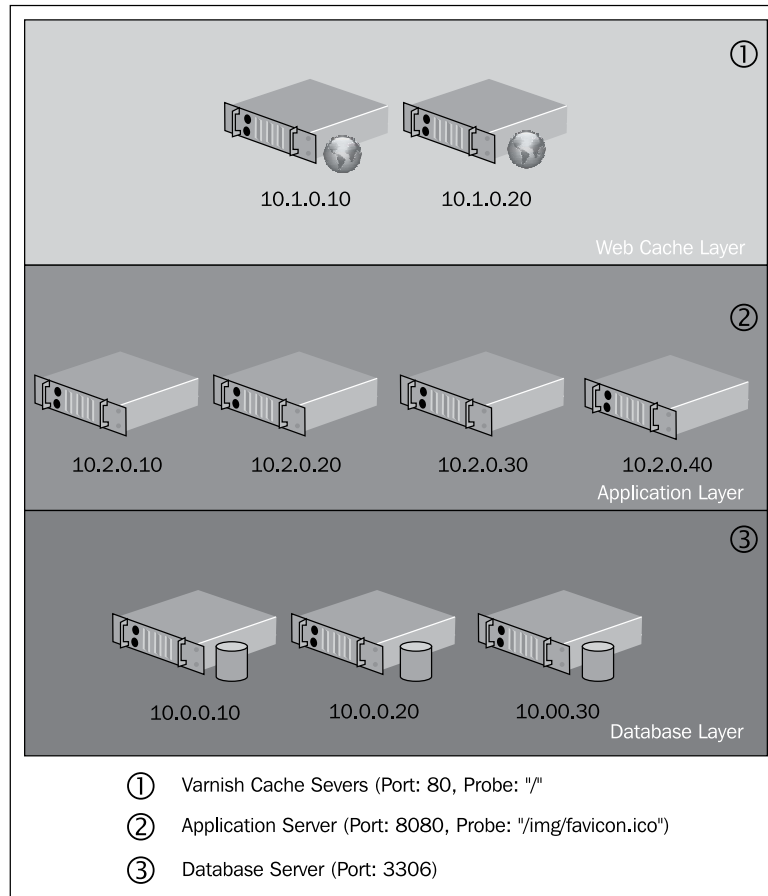
You can find more information about the persistent storage type on <https://www.varnish-cache.org/trac/wiki/ArchitecturePersistentStorage>.

Connecting to backend servers (Should know)

A backend server can be defined as any HTTP server from which Varnish Cache can request and fetch data. In this recipe, we will define our backend servers, probe those servers for their health status, and direct our clients' requests to the correct backend servers.

Getting ready

If you have a server architecture diagram, that's a good place to start listing all the required servers and grouping them, but you'll also need some technical data about those servers. You may find this information in a server monitoring diagram, where you will find the IP addresses, ports, and luckily a probing URL for health checks.



In our case, the main VCL configuration file `default.vcl` is located at `/etc/varnish` and defines the configuration that the Varnish Cache will use during the life cycle of the request, including the backend servers list.

How to do it...

1. Open the `default.vcl` file by using the following command:

```
# sudo vim /etc/varnish/default.vcl
```

2. A simple backend declaration would be:

```
backend server01 {  
    .host = "localhost";  
    .port = "8080";  
}
```

This small block of code indicates the name of the backend (`server01`), and also the hostname or IP, and which port to connect to.

3. Save the file and reload the configuration using the following command:

```
# sudo service varnish reload
```

At this point, Varnish will proxy every request to the first declared backend using its default VCL file. Give it a try and access a known URL (like the index of your website) through the Varnish Cache and make sure that the content is delivered as it would be without Varnish.

For testing purposes, this is an okay backend declaration, but we need to make sure that our backend servers are up and waiting for requests before we really start to direct web traffic to them.

4. Let's include a probing request to our backend:

```
backend website {  
    .host = "localhost";  
    .port = "8080";  
    .probe = {  
        .url = "/favicon.ico";  
        .timeout = 60ms;  
        .interval = 2s;  
        .window = 5;  
        .threshold = 3;  
    }  
}
```

Varnish will now probe the backend server using the provided URL with a timeout of 60 ms, every couple of seconds.

To determine if a backend is healthy, it will analyze the last five probes. If three of them result in 200 – OK, the backend is marked as Healthy and the requests are forwarded to this backend; if not, the backend is marked as Sick and will not receive any incoming requests until it's Healthy again.

5. Probe the backend servers that require additional information:

In case your backend server requires extra headers or has an HTTP basic authentication, you can change the probing from `URL` to `Request` and specify a raw HTTP request. When using the `Request` probe, you'll always need to provide a `Connection: close` header or it will not work. This is shown in the following code snippet:

```
backend api {
    .host = "localhost";
    .port = "8080";
    .probe = {
        .request =
            "GET /status HTTP/1.1"
            "Host: www.yourhostname.com"
            "Connection: close"
            "X-API-Key: e4d909c290d0fb1ca068ffaddf22cbd0"
            "Accept: application/json"
        .timeout = 60ms;
        .interval = 2s;
        .window = 5;
        .threshold = 3;
    }
}
```

6. Choose a backend server based on incoming data:

After declaring your backend servers, you can start directing the clients' requests. The most common way to choose which backend server will respond to a request is according to the incoming URL, as shown in the following code snippet:

```
vcl_recv {
    if ( req.url ~ "/api/" ) {
        set req.backend = api;
    } else {
        Set req.backend = website;
    }
}
```

Based on the preceding configuration, all requests that contain `/api/` (`www.yourdomain.com/api/`) in the URL will be sent to the backend named `api` and the others will reach the backend named `website`.

You can also pick the correct backend server, based on User-agent header, Client IP (geo-based), and pretty much every information that comes with the request.

How it works...

By probing your backend servers, you can automate the removal of a sick backend from your cluster, and by doing so, you avoid delivering a broken page to your customer. As soon as your backend starts to behave normally, Varnish will add it back to the cluster pool.

Directing requests to the appropriate backend server is a great way to make sure that every request reaches its destination, and gives you the flexibility to provide content based on the incoming data, such as a mobile device or an API request.

There's more...

If you have lots of servers to be declared as backend, you can declare probes as a separated configuration block and make reference to that block later at the backend specifications, avoiding repetition and improving the code's readability.

```
probe favicon {
    .url = "/favicon.ico";
    .timeout = 60ms;
    .interval = 2s;
    .window = 5;
    .threshold = 3;
}

probe robots {
    .url = "/robots.txt";
    .timeout = 60ms;
    .interval = 2s;
    .window = 5;
    .threshold = 3;
}

backend server01 {
    .host = "localhost";
    .port = "8080";
    .probe = favicon;
}

backend server02 {
    .host = "localhost";
    .port = "8080";
    .probe = robots;
}
```

The `server01` server will use the probe named `favicon`, and the `server02` server will use the probe named `robots`.

Load balance requests (Should know)

Balancing requests is a great way to share workload across the cluster pool and avoid overloading a single server instance, keeping the overall health of the system. There's also the possibility to direct VIP customers to a dedicated cluster pool, guaranteeing them the best user experience.

By using a director group, Varnish will manage and spread the incoming requests to those servers included in it. Having the servers constantly checked, Varnish can take care of the sick servers and maintain everything as if there was no problem at all.

Getting ready

There are six types of director groups to be configured: random, client, hash, round-robin, DNS, and fallback. While the random director is self-explanatory (randomly distributes requests), a DNS director can be used to spread to an entire network of servers. The hash director will always choose a backend based on the hash of the incoming URL and the fallback director can be used in emergency cases when servers behave oddly.

The two most common directors are the round-robin and client directors.

The round-robin can be used to spread requests one by one to the entire cluster of servers no matter what is requested, and the client director can be used to create a sticky-session based on unique information provided by the client, such as an IP address.

In this recipe we will create both the client and round-robin balancers to spread requests across application servers.

How to do it...

1. Group the backend servers to load balance requests by using the following code snippet:

```
director dr1 round-robin {
    { .backend = server01 }
    { .backend = server02 }
    { .backend = server03 }
    { .backend = server04 }
}
```

In the preceding example, we have declared that our director named `dr1` is a round-robin director, and inside this director there are four backend servers to be balanced. Backend servers `server01` to `server04` have already been configured earlier and this declaration is only referring to them.

2. Create a sticky-session pool of servers by using the following code snippet:

```
director dr1 client {  
    { .backend = server01 }  
    { .backend = server02 }  
    { .backend = server03 }  
    { .backend = server04 }  
}
```

At first, there's absolutely no difference from the `round-robin` declaration to the `client` one, but inside your `vcl_recv` subroutine, you'll need to specify what identifies a unique client. Varnish will use, as default, the client IP address, but if you have other services in front of your Varnish Cache (such as a firewall), you'll need to rewrite the value of the `client.identity` variable. In the following example, we'll use the `X-Forwarded-For` header to get the clients' real IP address.



The `X-Forwarded-For` header is used to maintain information lost in the proxying process. For more information visit <http://tools.ietf.org/html/draft-ietf-appsawg-http-forwarded-10>.

```
sub vcl_recv {  
    set client.identity = req.http.X-Forwarded-For;  
}
```

A sticky-session pool is necessary to direct clients to specific parts of your website that requires an HTTP session or authorization, such as shopping cart/checkout and login/logout pages, without breaking their session. Those parts of your website may be critical to your business, and having a sticky-session dedicated cluster can prioritize paying customers while the others are still browsing the products and will not interfere with the checkout process performance.

How it works...

By using directors to load balance the requests, we can obtain greater service availability and provide paying customers with an improved user experience.

There's more...

Sometimes it's not possible for all the servers to be identical inside a cluster. Some servers may have more available ram or more processors than others, and to balance requests based on the weakest server in the cluster is not the best way to solve this problem, since the higher end servers would be underused.

Weight-based load balancing improves the balance of the system by taking into account a pre-assigned weight for each server as shown in the following code snippet:

```
director dr1 client {  
    { .backend = server01 ; .weight=2; }  
    { .backend = server02 ; .weight=2; }  
    { .backend = server03 ; .weight=2; }  
    { .backend = server04 ; .weight=1; }  
}
```



Weighting the servers is only possible in the random or client directors.

The Varnish Configuration Language (Should know)

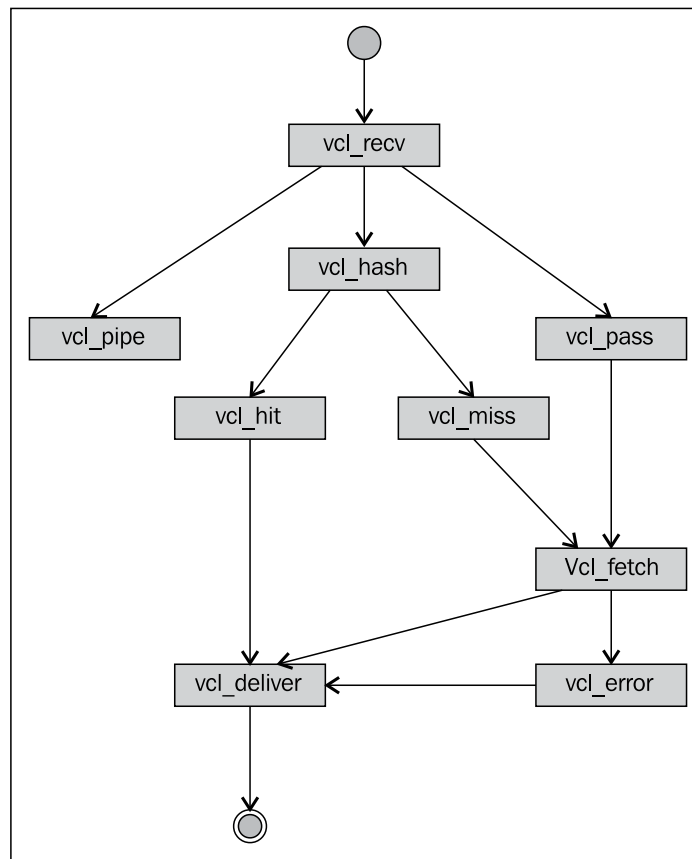
The **Varnish Configuration Language (VCL)** is a domain-specific language used to define the caching policy. It might seem a bit confusing at first to learn another DSL instead of using an already known language, but in the next recipe you will find out how easy it is to define how your cache should behave using the VCL.

Every written VCL code will be compiled to binary code when you start your Varnish Cache. So if you forget a semi-colon, for example, the code will not compile and the daemon will continue using the last compiled version of your configuration file, as long as you use the reload function from the startup script and don't restart your server.

Getting ready

The main configuration file `default.vcl` used to define the caching policy is entirely written in VCL and is located at `/etc/varnish/`.

Most of the configuration is written into subroutines and follows a pre-defined flow during the request and response phase.



How to do it...

1. Creating comments in the code:

You can use `//`, `#`, or `/*` your comment inside `*/` for creating a comment in your code:

```
// commented line
# commented line
```

2. Assignment and logical operators:

Assignments can be done with a single equals sign `=`.

```
set client.identity = req.http.x-forwarded-for;
```

Comparisons can be done with `==` for matching conditions or `!=` for not matching conditions.

```
if (req.restarts == 0)
if (req.request != "GET")
```

Logical operations can be done with `&&` for AND operations, and `||` for OR operations.

```
if (req.request != "GET"&&req.request != "HEAD")
```

3. Regular expressions:

Varnish uses the **Perl-compatible regular expressions (PCRE Regex)**.

The contains operator for validating regex is `~` (tilde) and the does not contain operator is `!~`.

```
if (req.url !~ "^/images/")
if (req.url ~ "\.(jpg|jpeg|png|gif)$")
```

4. VCL functions:

- ❑ `regsub()` and `regsuball()`: They work by changing a provided string whenever it matches a regex expression. The difference between the two functions is that the `regsub()` function replaces only the first match and the `regsuball()` function replaces all matching occurrences.

Both functions expect a `original_string`, `regex`, `substitute_string` and `format` as input:

```
regsub(req.url, "\?.*", "");
```

This expression will remove any character located after a `?` sign (including the question mark itself). Removing the parameters from the requested URL during the `vcl_hash` phase will help you avoid storing duplicated content (be careful not to end up serving the wrong content).

- ❑ `purge`: It is used to invalidate stale content, keeping the cache fresh. A good rule of thumb is to send an HTTP `PURGE` request to Varnish Cache whenever your backend receives an HTTP `POST` or `DELETE`.
- ❑ `ban()` and `ban_url()`: They create a filter, instructing if a cached object is supposed to be delivered or not. Adding a new filter to the ban list will not remove the content that is already cached—what it really does is exclude the matching cached objects from any subsequent requests, forcing a cache miss.

```
ban_url("\.xml$")
ban("req.http.host ~ " + yourdomain.com )
```



Too many entries in the ban list will consume extra CPU space.

- ❑ `return()`: It determines to which subroutine the request should proceed to. By calling `return(lookup)` method, we're instructing Varnish to stop executing the current subroutine and proceed to the `lookup` subroutine.


```
return(restart)
return(lookup)
return(pipe)
return(pass)
```
- ❑ `hash_data()`: It is responsible for setting the hash key used to store an object and it is only available inside the `vcl_hash` subroutine. The default value for the `hash_data` variable is `req.url` (requested URL), but it may not suit your needs. In a multi-domain website, a hash key conflict may occur, since the default `hash_data` for the homepage would be `/` in all domains. Concatenating the value of the `req.http.Host` variable would add the domain name to the hash key, making it unique.
- ❑ `error()`: It is used to interrupt a request or response whenever an error arises and an error page needs to be displayed. The first argument is the HTTP error code and the second is a string with the error code message.

```
if (!client.ip ~ purge) {
    error 405 "Not allowed.";
}
```

How it works...

All the VCL written code is translated to C language and compiled into a shared object, which will be linked to the server process when it is reloaded. Any coding mistake found during this phase, such as a missing semi-colon at the end of a line, will generate the following compiler error indicating the line and what the compiler was expecting:

Message from VCC-compiler:

Expected ';' got '}'

(program line 174), at

('input' Line 93 Pos 9)

}

-----#

Running VCC-compiler failed, exit 1

VCL compilation failed

In the preceding example, the VCC compiler informs that it was expecting a `;` (semi-colon) on line 93 and found a `}` (curly bracket) instead.

There's more...

To find out how Varnish is translating your VCL code to C language, run the daemon with the `-C` argument that will instruct the daemon to compile and print the C language code to the console.

```
# varnishd -C -f /etc/varnish/default.vcl
```

If you decide to insert an inline C code inside your VCL, analyzing the generated code will help you to debug it.

You will also find out that inside the generated C code the default VCL configuration is still present, even if you have deleted it. This is a self-defense mechanism which makes sure that a request always has a valid `return()` statement.

Handling HTTP request `vcl_recv` (Should know)

The `vcl_recv` routine is the first subroutine to be executed when a request comes in. At this point, you can normalize URL, add or subtract HTTP headers, strip or delete cookies, define which backend or director will respond to the request, control access, and much more.

First, we will take a look at the default `vcl_recv` subroutine and increment its behavior to suit our needs.

Getting ready

Open the `default.vcl` file at `/etc/varnish` and find the `vcl_recv` routine (sub `vcl_recv`).

The following block of codes are presented as an explanation of the default behavior and in the *How to do it...* section, we will modify them.

```
if (req.restarts == 0) {
    if (req.http.x-forwarded-for) {
        set req.http.X-Forwarded-For =
            req.http.X-Forwarded-For + ", " + client.ip;
    } else {
        set req.http.X-Forwarded-For = client.ip;
    }
}
```

The `req.restarts` object is an internal counter that indicates how many times the request was restarted (the prefix `req` indicates that this variable is of the type request). Restarting is commonly done when a backend server is not responding or an expected error arises, and by doing so, you can choose another backend server, rewrite the URL, or take other actions to prevent an error page.

This specific block of code is executed only when the request is not restarted, and it will append the client IP to an `X-Forwarded-For` header, if present. If not, Varnish will create the `X-Forwarded-For` header and assign the client IP to it.

```
if (req.request != "GET"&&
    req.request != "HEAD"&&
    req.request != "PUT"&&
    req.request != "POST"&&
    req.request != "TRACE"&&
    req.request != "OPTIONS"&&
    req.request != "DELETE") {
    /* Non-RFC2616 or CONNECT which is weird. */
    return (pipe);
}
```

If the incoming request is not a known HTTP method, Varnish will pipe it to the backend and let it handle the request.

```
if (req.request != "GET"&&req.request != "HEAD") {
    /* We only deal with GET and HEAD by default */
    return (pass);
}
```

Since delivering cached content can only be done in `GET` and `HEAD` HTTP methods, we need to pass the request that do not match to the backend servers.

```
if (req.http.Authorization || req.http.Cookie) {
    /* Not cacheable by default */
    return (pass);
}
```

If the request contains an authorization or cookies header, we should not try to look up for it in cache, since the content of this request is specific to the client.

```
return (lookup);
```

The final statement inside the `vcl_recv` subroutine instructs Varnish to look up the requested URL in cache. When the request does not match any of the previous conditions, which instructs Varnish not to look up for the content (like cookies, authorization, HTTP `POST`, and others), it will deliver a cached version of the requested content.

How to do it...

After understanding the default `vcl_recv` subroutine, we should identify the sections of our website that can be cached and the ones that cannot be cached. For some sections of the website, you still may want to deliver a cached version even if a user sends a cookie, for example, static content such as CSS, XML, and JPEG files.

1. Stripping cookies from requests:

```
if (req.url ~ "\.(png|gif|jpeg|jpg|ico|swf|css|js|txt|xml) "
|| req.url ~ "/static/"
|| req.url ~ "/images/") {
    unset req.http.cookie;
}
```

If the requested URL ends in any of the extensions (`.png`, `.jpeg`, `.ico`, and so on) listed in the condition, the cookies will be removed before the request continues through the VCL. The same behavior will happen for anything under the `static` and `images` directory.

Remember that Varnish will execute the code in a sequential way, so the first return statement that matches will be executed, and all the code after that will not be processed.

2. Define which backend or director will receive the request:

```
if(req.url ~ "/java/"
    set req.backend = java;
} else {
    set req.backend = php;
}
```

Every request that contains a `/java/` URL will hit the `java` director and everything else will hit the `php` director. This is a really basic example, and in reality you will probably use a regular expression to match sections of the requested URL.

3. Create security barriers to avoid unauthorized access:

```
if (req.url ~ "\.(conf|log|old|properties|tar|war)$") {
    error 403 "Forbidden";
}
```

If you want to, you can pass the request directly to the error subroutine `vcl_error` (with an HTTP error code and message), instead of using a return statement.

How it works...

Getting your subroutines right is one of the most important steps to get a good hit ratio, since the `vcl_recv` and `vcl_fetch` subroutines are probably where you will write 80 percent of your VCL code.

By stripping cookies of known static content, we get rid of requests that would always hit the backend for the same content like a header or footer, since the default behavior is to pass requests that contain cookies. Be extremely careful when stripping cookies, otherwise users may be locked outside your website if you remove sensitive data.

The `vcl_recv` subroutine is also the one where you control how balanced your cluster/directors will be, whenever a request passes through Varnish.

Handling HTTP request `vcl_hash` (Should know)

The `vcl_hash` subroutine is the subroutine that is executed after the `vcl_recv` subroutine. Its responsibility is to generate a hash that will become the key in the memory map of stored objects and play an important role in achieving a high hit ratio.

The default value for an object's key is its URL and it probably suits most of the cases. If you have too many SEO friendly URLs, and purging becomes a problem since those URLs are dynamically generated, using the `vcl_hash` subroutine to normalize these keys will help you remove stale content from cache.

Getting ready

This is the default action for the `vcl_hash` subroutine and the object's key will be of the URL + host format, in case an HTTP host header is present, or of the URL + server ip format, when it is not.

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

Whenever the `hash_data()` function is called, the provided value is appended to the current key.

How to do it...

1. Avoiding double caching for `www.yourdomain.com` and `yourdomain.com`:

```
sub vcl_hash {
    hash_data(req.url);
    if (req.http.host) {
        set req.http.host = regsub(req.http.host,
            "^www\.yourdomain\.com$", "yourdomain.com");
        hash_data(req.http.host);
    } else {
        hash_data(server.ip);
    }
    return (hash);
}
```

Adding the HTTP host normalization step before it is appended to the `hash_data()` function will prevent your cache from generating different keys for `www.yourdomain.com` and `www.yourdomain.com`, when the content is the same.

2. Removing domain name (host) from static files of multi-language websites:

```
if (req.url ~ "\.(png|gif|jpeg|jpg|ico|swf|css|js|txt|xml)"
{
    set req.http.X-HASH = regsub(req.url,
        ".*\.yourdomain\.[a-zA-Z]{2,3}\.[a-zA-Z]{2}", "");
}
hash_data(req.http.X-HASH);
```

Removing a domain name from the object's key can help you achieve a higher hit ratio by serving the same image for all web pages that speak the same language but have different domain names.

Creating a temporary header `http X-HASH` to rewrite and modify everything you need before passing to the `hash_data()` function can make your code more readable. By removing the domain name, the generated object's keys for `http://www.yourdomain.com/images/example.png` and `http://www.yourdomain.co.uk/images/example.png` will be the same.

3. Normalizing SEO URLs for easier purging:

```
if (req.url ~ "\.(html|htm)(\?[a-z0-9=]+)?$" {
    set req.http.X-HASH = regsub(req.http.X-HASH, "^/.*_",
        "/__");
}
hash_data(req.http.X-HASH);
```

Taking `http://yourdomain.com/a-long-seo-friendly-url-that-can-make-your-life-harder_3458.html` as an example of a SEO-friendly URL, removing this object from cache can become harder if the generated SEO URL is based on an updated field (as you will lose reference to the old key value), or if you have an asynchronous process that purges content, based on the timestamp of objects.

How it works...

Normalizing the object's key can make your life easier whenever you need to purge a content that is shared across multiple domains, or whenever you have a SEO friendly website in which it is necessary to update content as soon as it becomes stale.

Avoiding duplicated content will save extra memory space and provide you with a higher hit ratio, improving the overall performance of your website.

Take some time to familiarize yourself with your website contents, in order to identify files and sections that can benefit from a normalized key.

Handling HTTP request `vcl_pipe` and `vcl_pass` (Should know)

The main difference between these two subroutines is that the `vcl_pipe` subroutine will transmit the bytes back and forth after a pipe instruction is executed at the `vcl_recv` subroutine (and the subsequent VCL code will not be processed), no matter what is in it, while the `vcl_pass` subroutine will transmit the request to the backend without caching the generated response.

Both subroutines respect a `keep-alive` header. As long as the connection is still open, the `vcl_pipe` subroutine will keep transmitting bytes back and forth without analyzing any of the subsequent requests, preventing cached content from being delivered since the pipe is still active.

Piping or passing a request can be useful when users reach a protected section of the website.

Getting ready

The following is the default `vcl_pipe` subroutine:

```
sub vcl_pipe {
    # Note that only the first request to the backend will have
    # X-Forwarded-For set. If you use
    X-Forwarded-For and want to
    # have it set for all requests, make sure to have:
    # set bereq.http.connection = "close";
```

```
# here. It is not set by default as it might
# break some broken web applications, like IIS with NTLM
# authentication.
    return (pipe);
}
```

The default behavior is that if you do not set the HTTP connection header as `close`, a client might be piped once, and then all other subsequent requests made with a keep-alive header will use the same pipe, overloading your backend servers.

This is the default `vcl_pass` subroutine:

```
sub vcl_pass {
    return (pass);
}
```

The default `vcl_pass` behavior is pretty much self-explanatory. The request will be passed to the backend and the generated response will not be cached.

How to do it...

Add a connection header to piped requests:

```
sub vcl_pipe {
    set bereq.http.connection = "close";
    return (pipe);
}
```

Every piped request will be closed after it is processed.

How it works...

By piping requests, you can stream large objects, but you need to be careful or all other subsequent requests for that same client will also be piped.

Passing requests is the default action for protected or personalized sections of your website and requires no extra work.

Handling HTTP response `vcl_fetch` (Should know)

The `vcl_fetch` subroutine is the first subroutine to deal with the response phase and it plays an important role on caching policies and **Edge-side Include (ESI)**. When dealing with a legacy system that does not provide a `cache-control` header, you can hardcode a time to live (ttl) value to the content that should be cached.

While you can manipulate requests based on client-provided data using the `req.*` variable in the `vcl_recv` subroutine, you can do the same data manipulation in the `vcl_fetch` subroutine, but with data provided by a backend server using the `beresp.*` variable (`beresp` = backend response).



For more information about edge-side include visit <http://www.w3.org/TR/esi-lang>.

Getting ready

First we will take a look at the default `vcl_fetch` subroutine:

```
sub vcl_fetch {
    if (beresp.ttl <= 0s ||
        beresp.http.Set-Cookie ||
        beresp.http.Vary == "*") {
        /*
            * Mark as "Hit-For-Pass" for the next 2
minutes
            */
        set beresp.ttl = 120 s;
        return (hit_for_pass);
    }
    return (deliver);
}
```

The default `vcl_fetch` behavior will not cache the response if your backend server provides a zero or negative ttl value, a `Set-cookie` header, or a `Vary` header. Instead, Varnish will cache a dummy object that instructs the next requests for this URL to be passed for the next two minutes. This is called hit-for-pass.

How to do it...

1. Overriding the default time to live of a cached object:

```
if ( req.url ~ "\.(png|gif|jpeg|jpg|ico|css|js|txt|xml)
    (\?[a-z0-9=]+)?$" ) {
    set beresp.ttl = 1d;
}
```

Using the `set beresp.ttl = 1d` instruction, our static files will be stored in cache for one day. If our backend server provides an HTTP `cache-control` or `expires` header with a different time frame, we will override it with the `set` command.

2. Stripping cookies for static content:

```
if ( req.url ~ "/static/") {
    set beresp.ttl = 30m;
    unset beresp.http.set-cookie;
}
```

Removing the HTTP `set-cookie` header from the response allows us to sanitize the object before inserting it into memory.

3. Restart requests that failed:

```
if ( beresp.status >= 500 && req.request != "POST") {
    return(restart);
}
```

In case the backend server returns a 500+ HTTP error code (server-side error) and the original request was not an HTTP `POST`, we will restart it and try a different backend. Restarting will take the request back to the `vcl_recv` subroutine.

To do so, we also need to tweak the `vcl_recv` subroutine so that the restarted request can pick another backend instead of our original failed server.

```
sub vcl_recv {
    if (req.restarts == 0) {
        # Try the director first.
        set req.backend = director1;
    } else if (req.restarts == 1) {
        # Director has failed and we will try the backend 1.
        set req.backend = b1;
    } else if (req.restarts == 2) {
        # Backend 1 has failed. Try backend 2.
        set req.backend = b2;
    } else {
        # All backend servers have failed. Go to error page.
        error 503 "Service unavailable";
    }
}
```

4. Inspecting why the response was not cached:

Sending a debug header alongside the response can help you understand the behavior of the cache and why a specific content was not cached.

```
sub vcl_fetch {
    if (beresp.ttl <= 0s) {
        # Cannot cache. Backend provided an expired TTL
        set beresp.http.X-Cacheable = "NO:ExpiredTTL";
    } elsif (req.http.Cookie) {
```

```

    # Presence of cookies.
    set beresp.http.X-Cacheable = "NO:Cookies";
  } elseif (beresp.http.Cache-Control ~ "private") {
    # Cache-control is private
    set beresp.http.X-Cacheable = "NO:Cache-
      Control=private";
  } else {
    set beresp.http.X-Cacheable = "YES";
  }
  return(deliver);
}

```

The preceding code will check for the presence of an expired time to live, cookies, or if the Cache-control header instructed that the content is private.

How it works...

Stripping cookies from a response will help you clean up data that should not be stored along with the object in cache, but it may be risky to do so, since cookies are used to authenticate users or track user steps. Be extra sure that the user experience will not be affected.

It is also possible to restart a request that would otherwise end up in an error inside the `vcl_error` subroutine, but there's no need to take that extra step if you can restart it as soon as the backend server throws an error. This is not a bulletproof method, but it can most certainly help you when a bad deploy happens in one of the cluster servers.

Handling HTTP response `vcl_deliver` (Should know)

The `vcl_deliver` subroutine is the last step before a response returns to the client and can be used for server obfuscation, adding debug headers, and a last overall headers' clean up.

Because the `vcl_deliver` subroutine is executed after the object is placed into cache, all manipulation that happens inside the `vcl_deliver` subroutine will not be persisted.

Getting ready

This is the default `vcl_deliver` behavior:

```

sub vcl_deliver {
    return (deliver);
}

```

As you can see, the default `vcl_deliver` subroutine is very straightforward and in fact it does not modify anything.

How to do it...

1. Remove the server name for obfuscating purposes:

```
sub vcl_deliver {
    unset resp.http.Server;
    return (deliver);
}
```

Obfuscating the HTTP `Server` header is a good way to avoid exposing what type of server and version you are using behind your Varnish Cache.

2. Removing extra headers added by Varnish:

```
sub vcl_deliver {
    unset resp.http.Server;
    unset resp.http.Via;
    unset resp.http.Age;
    unset resp.http.X-Varnish;
    return (deliver);
}
```

The HTTP `Age` header provides information about how long this object has been cached, while the HTTP `Via` header informs who delivered the response. The `X-Varnish` header returns two values: the first one is the request ID that originated the cached object, and the second one is the present request ID.

There is absolutely no need to remove those headers, but it is a good way to not give away unnecessary information.

3. Add cache hit or miss debug headers:

```
sub vcl_deliver {
    if (obj.hits > 0) {
        set resp.http.X-Cache = "HIT";
    } else {
        set resp.http.X-Cache = "MISS";
    }
    unset resp.http.Via;
    unset resp.http.Age;
    unset resp.http.X-Varnish;
    unset resp.http.Server;
    return (deliver);
}
```

Inside the `vcl_deliver` subroutine, you can read the object's `hit` counter variable to determine if the object is being served from cache or by the backend.

Adding a debug header in order to know if the content was served from cache helps you to identify objects that should be served from cache, but are not.

Handling HTTP response vcl_error (Should know)

The `vcl_error` subroutine handles odd behaviors from backend servers, and can also be used when you want to deny access to a request or redirect requests to a new location.

Another good use of the `vcl_error` subroutine is to deliver a maintenance page while you are working on the backstage, rolling out patches, or deploying a new version of your website (in case you cannot have multiple versions deployed at the same time).

Getting ready

We will take a look at the default `vcl_error` subroutine:

```
sub vcl_error {
    set obj.http.Content-Type = "text/html; charset=utf-8";
    set obj.http.Retry-After = "5";
    synthetic {
        <?xml version="1.0" encoding="utf-8"?>
        <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
        <html>
            <head>
                <title>"} + obj.status + " " + obj.response + {"</title>
            </head>
            <body>
                <h1>Error "} + obj.status + " " + obj.response + {"</h1>
                <p>"} + obj.response + {"</p>
                <h3>Guru Meditation:</h3>
                <p>XID: "} + req.xid + {"</p>
                <hr>
                <p>Varnish cache server</p>
            </body>
        </html>
    };
    return (deliver);
}
```

The default `vcl_error` subroutine will generate a simple HTML page informing the status of the response and the ID of the request.

Custom error pages can be declared inside the synthetic block.

How to do it...

1. Maintenance page:

Whenever you need to take the entire website offline (as a last resort), you can redirect users straight from the request phase (`vcl_recv`) to the `vcl_error` subroutine and deliver a static page that warns the users about the scheduled maintenance.

```
sub vcl_recv {
    error 503 "Service unavailable";
}
sub vcl_error{
    if ( obj.status == 503 ) {
        set obj.http.Content-Type = "text/html; charset=utf-8";
        set obj.http.Retry-After = "5";
        synthetic {"
            ### Insert here the HTML for the maintenance page ###
        "}
    }
}
```

2. Redirecting users:

Inside the `vcl_error` subroutine you can read and write to requests, objects, and responses variables. In case of an unexpected failure, you can redirect customers to an **Oops! We're sorry** friendly page or even to the website home page.

```
sub vcl_error {
    if (obj.status >= 500) {
        set obj.http.location = "http://www.yourwebsite.com";
        set obj.status = 302;
        return (deliver);
    }
}
```

Keep in mind that the 301 HTTP code means Moved Permanently, so dealing with redirections based on a system failure cannot be assigned as a 301 status. A 302 HTTP code informs that the resource was temporarily moved.

How it works...

Redirecting customers to a friendly error page is a good way to avoid users from having a bad experience, therefore favoring a competitor's website.

Caching static content (Should know)

A static web page can be defined as a page that is pre-built and delivered exactly the same way every time it is loaded. Even if the content of a page is updated from time to time (let's say every 10 to 15 minutes) with the latest news or products, you can still cache that page for a smaller period and benefit from not having to process that same response for every customer inside that time frame.

Other cacheable static contents that can make your website faster are the CSS, JavaScript, and image files which will probably be cached for a longer period of time than your HTML pages.

Getting ready

First, you need to identify your static files or sections and define a time to live based on how long you expect a specific type of file to change its content.

Creating a list of files and sections helps you grouping them under a single VCL condition (rule) based on their expected ttl:

File type	Time period
Image files (JPG, PNG, GIF)	1 week
JavaScript	1 hour
CSS	1 hour

The best place to define your ttl caching policies is inside the `vcl_fetch` subroutine because it is the last step before an object is inserted to cache.

How to do it...

1. Define a time to live based on file type:

```
sub vcl_fetch {
    if ( req.url ~ " \.(png|jpg|gif)$" ) {
        set beresp.ttl = 1w;
        unset beresp.http.Set-Cookie;
    }
    if ( req.url ~ " \.(css|js)$" ) {
        set beresp.ttl = 1h;
        unset beresp.http.Set-Cookie;
    }
    if ( req.url ~ " \.(html)$" ) {
        set beresp.ttl = 15m;
        unset beresp.http.Set-Cookie;
    }
}
```

As you have probably already noticed, Varnish takes a numeric argument followed by a single letter, which identifies a scale of time:

S = seconds

H = hours

D = days

W = weeks

2. Disabling cache:

```
sub vcl_fetch {  
    if ( req.url ~ "/donotcache\.html" ) {  
        set beresp.ttl = 0s;  
    }  
}
```

Setting the ttl to zero seconds will create an already expired object, forcing it not to be cached.

How it works...

Avoiding duplicated workload is the first step to a faster website, and achieving a higher hit ratio for your static content is where you have to primarily focus before moving to more complex caching policies.

Make sure your backend is not already defining a caching policy with `cache-control` or `expire` headers, as you probably will be better off obeying those policies.

Un-setting cookies before the object is placed into memory is mandatory or else you will end up caching unnecessary and sensitive information.

Cookies, sessions, and authorization (Become an expert)

There are many ways to identify unique users in our website. In this recipe, we will cover the three most used ones: cookies, sessions, and authorization. Whenever one of them occurs, you must be extremely careful with the request and generated response since they contain user-specific data and cannot be delivered to a different user.

Caching user-specific content requires an extra step, since only the same user will be able to access that cached object. Delivering user-specific content to someone else can lead to session hijacking, user's private information leakage, and many other security problems.

Getting ready

An HTTP cookie, also referred to as browser cookie, is the result of a `set-cookie` response header and it is used to store small pieces of data on the client side (browser) for later usage on subsequent requests. The top reason for using cookies is to identify a unique user and use the stored information to generate personalized content or to track its steps.

HTTP session is generated on the server side and only the ID of that interaction is sent to the client in the form of a cookie. A session is used to keep all the information on the server side, and the user only handles the session ID (token), unlike what happens with a regular cookie. Session tokens are often specific to programming languages.

HTTP authorization is the most basic kind of HTTP security that can be implemented in a website (often called HTTP Basic Auth), and it is executed via an `authorization` header. HTTP authorization will often encrypt the username and password in a base 64 fashion.

How to do it...

1. Avoiding cache for user-specific requests:

```
sub vcl_recv {
    if (req.http.Authorization || req.http.Cookie) {
        return (pass);
    }
}
```

Since sessions are stored inside cookies, you will only need to check for the presence of a session as a URL parameter, like the `JSESSIONID` (for Java systems).

2. Caching user specific information:

If you need to create a per-user cache, the following example of code will set the cookie along with the original `hash_data`. Caching responses with cookies will often lead to more problems than solutions.

```
sub vcl_recv {
    if (req.http.Cookie) {
        return (lookup);
    }
}
```

```
sub vcl_hash {
    hash_data(req.url);
    if ( req.http.Cookie ~ "cookie_set_for_unique_users" ) {
        hash_data(req.http.cookie);
    }
}
```

This is actually a basic example on how to deal with requests that contain cookies and need to be cached. You should write your own regex (using the `regsub` function) to extract only the value of the cookie that you will use to append to the default object's key.

How it works...

Storing user-specific content in cache can be helpful to deliver a faster response, but it is not recommended since the non-authenticated users represent the biggest portion of the requests, therefore, the authenticated users will not have trouble if they do not hit the cache at all.

Unless you have a website that requires all users (or at least half of them) to be logged in, it is better to let the requests pass to backend servers and avoid the trouble of leaking private information and other security concerns.

HTTP cache headers (Should know)

The HTTP is a long-established protocol for exchanging information between clients and servers, and the headers are an important part of that communication. HTTP headers define what the client is requesting and how they expect it to be responded to by the server.

Along with a request, you will probably find a handful of headers such as `accept`, `accept-encoding`, `keep-alive`, `host`, and many others. In a `GET` method, the request will ask for information about a specific resource and attach headers to allow the server to know how the response should be formed.

On the response phase of the communication, the server will also attach server-specific headers to let the client know how it should treat the response and, if instructed, store it (browser cache).

Getting ready

In this section, we will focus primarily on the HTTP header called `cache-control` which is covered in the section 13 of the Hypertext Transfer Protocol Version 1.1 (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html>).

How to do it...

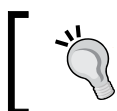
The HTTP protocol Version 1.1 introduced the `cache-control` response header as an alternative to the `expires` header implemented in HTTP 1.0. The main difference between them is that `expires` takes a date value and the `cache-control` can receive an age value.

Expires: `Fri, 30 Oct 1998 14:19:41 GMT`

Cache-Control: `max-age=3600`

1. The `cache-control` header can also indicate what level of cache should be done for that response. This type of control is passed as a directive and can assume the following values:
 - ❑ `public`: Any type of cache mechanism (server, proxy, or browser) can cache the content of that response.
 - ❑ `private`: Indicates that the response (or part of it) is specific to a single client and should not be cached by a "shared" cache mechanism.
 - ❑ `no-cache`: Forces a validation request to the origin server before delivering the cached copy to the user. It is primarily used for authentication when a cached copy of the response cannot be delivered to an unauthorized client.
 - ❑ `no-store`: The response cannot be cached at all.
2. The expiration attributes (ttl) of the `cache-control` header are:
 - ❑ `max-age`: Defines for what period of time the response will be cached.
 - ❑ `s-maxage`: Exactly the same as `max-age`, but it only applies to proxy caches.

Opposite to the `cache-control` header, the `expires` header receives an HTTP-date (RFC 1123) as value that indicates until when the content of the response is considered valid. After it expires, a new representation will be requested from the origin server.



The HTTP-date specification can be found at section 3.3.1 of the HTTP protocol (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html>).

How it works...

The HTTP protocol is the foundation in which Varnish relies and includes a number of elements to make caching policies easier. The best approach to setting different times for cached objects (when using a proxy-cache) is to code the time to live policies inside your application server, by sending a `cache-control` header alongside with the response and thus avoiding to hardcode the ttl inside the VCL code. In case you are dealing with a legacy system that is no longer maintained, you will probably need to hardcode the ttl to create a cache object.

Varnish will respect the cache headers values unless it is told not to. So whenever a content should be cached and it is not, check the response headers for any indication of a `no-store`, `no-cache`, or maybe an already expired time value.

Invalidating cached content (Should know)

Up until now, we have only seen ways of caching contents of your website, but delivering stale data to clients is undesirable when an updated version of that content has already been deployed—this is actually worst than not having a cache at all.

A simple example of bad content (stale) is the representation of a product that is on sale, and after the discount expires, the server keeps showing its discounted value. Since you would not cache a sensitive portion of your website, like the shopping cart, the user would see a discount price at the product detail page and a higher price at the shopping cart.

Getting ready

There are actually two ways of invalidating cached contents, and it is very important that you understand the difference between them.

- ▶ **Purge:** It is used to remove a single object from cache and must be implemented inside both the `vcl_hit` and `vcl_miss` subroutines. A single object can contain multiple cached versions if the content is delivered with a `vary` header, and the purge method will remove all existing variations.
- ▶ **Ban:** It is useful for removing large amounts of contents at the same time by using regular expressions as `match`. In case you need to ban all JPEGs (`.jpg`) or any other regex match, the ban method is faster than the purge method because the latter will invalidate them one by one, thus taking a lot longer to process.

Be aware that while the purge will remove the content from cache, the ban method does not remove it right away—what it actually does is create a ban list that is checked every time a request comes, trying to find out if the content should be delivered from cache or if a newer version should be fetched from the backend server. At this point, you must probably have concluded that this will generate extra workload and a big ban list filter may increase the load on your server.

How to do it...

1. Purge:

When using any method of invalidation, it is always a good idea to have an **Access Control List (ACL)** with the authorized hosts.

```
acl purge {  
    "localhost";  
}
```

```
"10.1.0.0"/24;  
"172.16.11.0"/23;  
}
```

In the preceding example, we are adding `localhost` and all hosts from the `10.1.0.0` and `172.16.11.0` networks to an ACL called `purge` (you can name the ACL any way you want).

```
sub vcl_hit {  
    if (req.request == "PURGE") {  
        if (!client.ip ~ purge) {  
            error 405 "Not allowed.";  
        } else {  
            purge;  
            error 200 "Purged.";  
        }  
    }  
    return (deliver);  
}  
  
sub vcl_miss {  
    if (req.request == "PURGE") {  
        if (!client.ip ~ purge) {  
            error 405 "Not allowed.";  
        } else {  
            purge;  
            error 200 "Not in cache.";  
        }  
    }  
    return (fetch);  
}
```

Even though the `PURGE` method is not present in the HTTP protocol, it is actually a `GET` method with a different name that is used to direct requests to the `purge` method inside the VCL.

The exact requested object might not generate a hit and a variant of that object might be in cache, so using the `purge` method inside the `vcl_miss` subroutine is necessary to remove all variants.

2. Ban:

Ban can be done through VCL code:

```
sub vcl_recv {  
    if (req.request == "BAN") {  
        if (!client.ip ~ purge) {  
            error 405 "Not allowed.";  
        }  
    }  
}
```



```
    }  
    ban("req.http.host == " + req.http.host + "&&req.url ==  
        " + req.url);  
    error 200 "Banned";  
  }  
}
```

But, it can also can be done through the varnish CLI:

```
ban req.url ~ "\.jpg$"
```

How it works...

While you may increase the ttl of cached objects in hope that it will improve the chances of a cache hit, serving stale content is bad and should be avoided at any cost. Implementing a cache invalidation policy is as important as getting objects into cache and should be treated as a top priority.

You should always purge content whenever your application server receives an update to its own entities. Since this behavior is not always present in legacy systems, you may need to remove stale data using the varnish CLI.

Compressing the response (Become an expert)

HTTP compression can be described as a way to reduce the transferred data between servers and client. Reducing the amount of transmitted data will result in a faster loading website, and in the new cloud datacenter era, it may even reduce your network usage costs.

Compression can be performed mainly on text content such as HTML, CSS, JS, XML files, but it does not mean that other type of files cannot be compressed. A common mistake is to compress files that are already natively compressed, such as a PNG image file. This odd behavior will only reduce performance as the compression and decompression processes will actually consume more time and will not result in a smaller file.

Getting ready

Only after Version 3.0 of the Varnish Cache, the gzip compression method is possible and is definitely encouraged.

Varnish default behavior is to compress the response before delivering it to the client, searching for the presence of the `accept-encoding` header in the request. You can change this behavior by setting the `http_gzip_support` parameter in the Varnish daemon.

Even if the `accept-encoding` header is present in the request, this does not guarantee that the cached object will be stored as a compressed representation. In the following section, we will compress objects before inserting them into cache.

How to do it...

1. Compressing before storing:

Setting the `do_gzip` variable to `true` inside the `vcl_fetch` subroutine will enable the gzip compression before the content is stored.

```
sub vcl_fetch {
    if (beresp.http.content-type ~ "text") {
        set beresp.do_gzip = true;
    }
}
```

2. Compressing specific content:

```
sub vcl_fetch {
    if ( req.url ~ "\.(html|htm|css|js|txt|xml) (\?[a-z0-9=]+)?$" ) {
        set beresp.do_gzip = true;
    }
}
```

You should not try to compress files that are already natively compressed such as JPEGs, MP3s, and so on, avoiding unnecessary workload.

How it works...

Compressing the response will result in a smaller network footprint, a faster loading time of pages, and in reduced costs if your datacenter charges you for network bandwidth usage.

Since most of the search engines will add relevance to faster websites, consider this tool as one of the most important implementation inside your VCL code.

Monitoring the hit ratio (Become an expert)

Varnish default installation comes with many secondary tools that can make the debugging and analysis tasks easier and faster, including `varnishlog` (to access the shared memory log), `varnishncsa` (to generate an access log in the apache common format), `varnishhist` (to create a histogram with hit-miss along with time), `varnishadm` (for administrative interface), and the `varnishstat` that we will cover in this recipe.

Analyzing the HTTP headers contained inside the requests and responses will solve most of the problems you may encounter while debugging. An HTTP header viewer such as the Firefox add-on **Live HTTP Headers** will be very helpful to sort out why a response is not being cached, but some other type of bugs will take extra investigation.

Getting ready

Before trying to improve your cache by reducing the number of repeated objects in memory and refactoring your VCL code to make it slimmer, take a minute to see how well you cache is performing live.

If you Varnish instance has a low hit ratio percentage, you may need to implement more aggressive caching policies by forcing to cache static content that is not being stored due to a misconfigured header.

How to do it...

1. Open the `varnishstat` monitoring tool.

```
# varnishstat
```

2. Type in your server console screen the preceding command to execute the Varnish statistics tool.

You will be presented with a screen full of statistics and we will go through the most important ones here in this section.

```
0+00:01:09
Hitrate ratio:      10      17      17
Hitrate avg:       0.7770  0.7731  0.7731
```

The upper left part of the `varnishstat` shows the uptime of the server (Days+Hours:Minutes:Seconds).

The **Hitrate ratio** indicates the time frame, in seconds, of the collected data.

The **Hitrate avg** numbers show the percentage of hits (multiply by 100) according to the time frame right above them.

In this example, the hit ratio average was 77 percent for the last 10 seconds, as it also was in the remaining time frames.

The following data is formatted as:

Raw data / realtime (per second) / since boot (per second)

```
865    8.99  12.54 client_conn - Client connections accepted
13568 199.71 196.64 client_req - Client requests received
```

8.99 connections accepted per second.

199.71 requests being made.

The preceding lines show the client connections and requests. The client connections are expected to be lower than the requests, since `KeepAlive` headers were sent with the connections.

```
9886 162.76 143.28 cache_hit - Cache hits
122 1.00 1.77 cache_hitpass - Cache hits for pass
3647 33.95 52.86 cache_miss - Cache misses
```

162.76 cache hits per second.

33.95 cache misses per second.

A high cache miss counter is the first sign of a bad caching policy that needs revision. There is not a magic number when it comes to hit ratio average, but if your cache is not effective, there is no point in caching.

```
85 0.00 1.23 backend_conn - Backend conn. success
4239 37.94 61.43 backend_reuse - Backend conn. reuses
```

0.00 connections to the backend.

37.94 connections reused to the backend.

3. Do not be scared if you find out that in the last second no connections were made to the backend servers. This is a good sign of connections being reused, which avoid unnecessary handshake between servers. We are aiming for a high backend reuse.
4. There are many other counters to retrieve information from the statistics about the health of your Varnish instance and whether there is a problem with it or not. You should pay close attention to the hit ratio and how well Varnish is connecting to the backend servers.



To quit the `varnishstat` interface just hit `Ctrl + C`.

How it works...

Monitoring the hit ratio average, the backend connections and other information after a new VCL has been deployed is the key to identifying early problems before it crashes and a server restart is required.

The smallest mistake while coding a personalized hash key or misplacing a return statement can ruin your cache and make your backend servers go down with the increased and unexpected load.



Thank you for buying
Instant Varnish Cache How-to

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

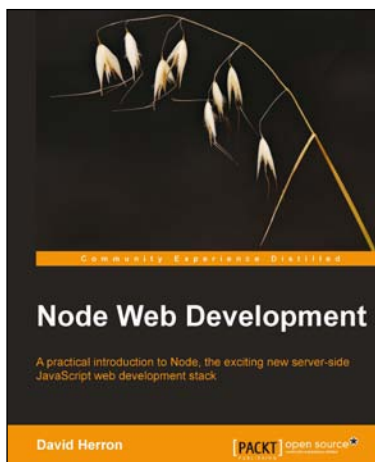
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



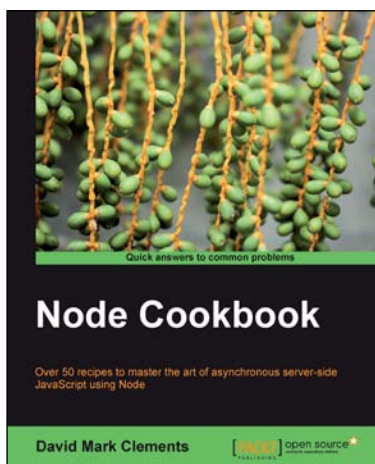
Node Web Development

ISBN: 978-1-84951-514-6

Paperback: 172 pages

A practical introduction to Node, the exciting new server-side JavaScript web development stack

1. Go from nothing to a database-backed web application in no time at all
2. Get started quickly with Node and discover that JavaScript is not just for browsers anymore
3. An introduction to server-side JavaScript with Node, the Connect and Express frameworks, and using SQL or MongoDB database back-end



Node Cookbook

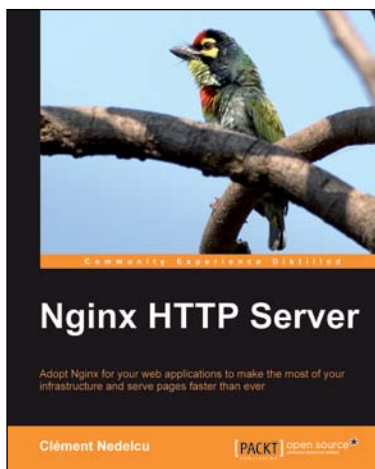
ISBN: 978-1-84951-718-8

Paperback: 342 pages

Over 50 recipes to master the art asynchronous server-side JavaScript using Node

1. Packed with practical recipes taking you from the basics to extending Node with your own modules
2. Create your own web server to see Node's features in action
3. Work with JSON, XML, web sockets, and make the most of asynchronous programming

Please check www.PacktPub.com for information on our titles



Nginx HTTP Server

ISBN: 978-1-84951-086-8

Paperback: 348 pages

Adopt Nginx for your web applications to make the most of your infrastructure and serve pages faster than ever

1. Get started with Nginx to serve websites faster and safer
2. Learn to configure your servers and virtual hosts efficiently
3. Set up Nginx to work with PHP and other applications via FastCGI
4. Explore possible interactions between Nginx and Apache to get the best of both worlds



CoffeeScript Programming with jQuery, Rails, and Node.js

ISBN: 978-1-84951-958-8

Paperback: 140 pages

Learn CoffeeScript programming with the three most popular web technologies around

1. Learn CoffeeScript, a small and elegant language that compiles to JavaScript and will make your life as a web developer better
2. Explore the syntax of the language and see how it improves and enhances JavaScript
3. Build three example applications in CoffeeScript step by step

Please check www.PacktPub.com for information on our titles