

Tim Butler

NGINX

Cookbook

Make the most of your web server



Packt>

NGINX Cookbook

Make the most of your web server

Tim Butler



BIRMINGHAM - MUMBAI

NGINX Cookbook

Copyright © 2017 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: August 2017

Production reference: 1300817

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

ISBN 978-1-78646-617-4

www.packtpub.com

Credits

Author Tim Butler	Copy Editor Tom Jacob
Reviewers Claudio Borges Jesse Lawson	Project Coordinator Judie Jose
Commissioning Editor Pratik Shah	Proofreader Safis Editing
Acquisition Editors Prachi Bisht Subho Gupta	Indexer Tejal Daruwale Soni
Content Development Editor Monika Sangwan	Graphics Kirk D'Penha
Technical Editor Bhagyashree Rai	Production Coordinator Arvindkumar Gupta

About the Author

Tim Butler is currently working in the web hosting industry and has nearly 20 years of experience. He currently maintains hyper-converged storage/compute platforms and is an architect of high throughput web logging and monitoring solutions.

You can follow him on Twitter using his Twitter handle, [@timbutler](#), where he (infrequently) posts about hosting, virtualization, NGINX, containers, and a few other hobbies.

About the Reviewers

Claudio Borges is a systems engineer with a computer science degree and over 15 years of experience in Linux/BSD. He has strong knowledge of systems administration and deployment with extensive experience in developing tools to automate systems and tasks.

I am grateful to all of those with whom I have had the pleasure to work during this project. I am especially indebted to the Packt team for giving me this opportunity.

This work would not have been possible without my family's support—my loving and supportive wife, Rose and my wonderful daughter, Victoria, who provide unending inspiration.

Jesse Lawson is a PhD student at Northcentral University and an information systems administrator at Butte College in Oroville, CA. His research addresses two primary areas: consumer psychology in higher education administration, and data science and analytics in the social sciences. His dissertation explores how machine learning models compare to human-generated processes of predicting college student outcomes, and what it means if people are better at predicting student dropout behavior than algorithms. He is the author of the bestselling book *Data Science in Higher Education*, published by *CreateSpace Independent Publishing Platform*; he is a technical reviewer for *Packt Publishing*; and a former technical reviewer for the *International Journal of Computer Science and Innovation*.

I'd like to thank my wife, Sami, for her uncompromising support to my pursuits.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. 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.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786466171>.

If you'd like to join our team of regular reviewers, you can email us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

What this book covers

What you need for this book

Who this book is for

Sections

Getting ready

How to do it...

How it works...

There's more...

See also

Conventions

Reader feedback

Customer support

Downloading the example code

Downloading the color images of this book

Errata

Piracy

Questions

1. Let's Get Started

Introduction

A quick installation guide

How to do it...

Packages – RHEL/CentOS

Packages – Debian/Ubuntu

Compiling from scratch
Debian/Ubuntu

CentOS/RHEL

Testing

How to do it...

There's more...

Configuring NGINX
How to do it...

How it works...

Enabling modules
How to do it...

See also

Deploying a basic site
How to do it...

How it works...

Basic monitoring
How to do it...

How it works...

Real-time statistics
How to do it...

How it works...

See also

2. Common PHP Scenarios

Introduction

Configuring NGINX for WordPress
Getting ready

How to do it...

How it works...

[There's more...](#)

[WordPress multisite with NGINX](#)

[How to do it...](#)

[Subdomains](#)

[See also](#)

[Running Drupal using NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Using NGINX with MediaWiki](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

[Using Magento with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Configuring NGINX for Joomla](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

3. Common Frameworks

[Introduction](#)

[Setting up Django with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works....](#)

[See also](#)

[Setting up NGINX with Express](#)

[Getting ready](#)

[How to do it...](#)

[How it works....](#)

[See also](#)

[Running Ruby on Rails with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Easy Flask with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Laravel via NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Meteor applications with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[High speed Beego with NGINX](#)

[Getting ready](#)

[How to do it...](#)

4. All About SSLs

[Introduction](#)

[Basic SSL certificates](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Enabling HTTP/2 on NGINX](#)

[How to do it...](#)

[See also](#)

[Configuring HSTS in NGINX](#)

[How to do it...](#)

[There's more...](#)

[Easy SSL certificates with Let's Encrypt](#)

[How to do it...](#)

[See also](#)

[Making NGINX PCI DSS compliant](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[OCSP stapling with NGINX](#)

[How to do it...](#)

[See also](#)

[Achieving full A+ Qualys rating](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

5. Logging

[Introduction](#)

[Logging to syslog](#)

[How to do it...](#)

[Remote syslog](#)

[See also](#)

[Customizing web access logs](#)

[How to do it...](#)

[See also](#)

[Virtual host log format](#)

[How to do it...](#)

[Application focused logging](#)

[How to do it...](#)

[Logging TLS mode and cipher information](#)

[How to do it...](#)

[Logging POST data](#)

[How to do it...](#)

[Conditional logging](#)

[How to do it...](#)

[Using the Elastic Stack](#)

[How to do it...](#)

[Elasticsearch](#)

[Logstash](#)

[Kibana](#)

[See also](#)

6. Rewrites

[Introduction](#)

[Redirecting non-www to www-based sites](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

Redirecting to a new domain

How to do it...

How it works...

There's more...

Blocking malicious user agents

How to do it...

How it works...

There's more...

Redirecting all calls to HTTPS to secure your site

How to do it...

There's more...

See also

Redirecting pages and directories

How to do it...

Single page redirect

Full directory redirect

How it works...

See also

Redirecting 404 errors through a search page

How to do it...

How it works...

7. Reverse Proxy

Introduction

Configuring NGINX as a simple reverse proxy

Getting ready

How to do it...

How it works...

There's more...

[See also](#)

[Content caching with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Monitoring cache status](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Microcaching](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Serving from cache when your backend is down](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[SSL termination proxy](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

[Rate limiting](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

8. Load Balancing

[Introduction](#)

[Basic balancing techniques](#)

[Round robin load balancing](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Least connected load balancing](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Hash-based load balancing](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Testing and debugging NGINX load balancing](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[See also](#)

[TCP / application load balancing](#)
[How to do it...](#)

[How it works...](#)
[Easy testing](#)

[There's more...](#)

[See also](#)

[NGINX as an SMTP load balancer](#)
[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

9. Advanced Features

[Introduction](#)

[Authentication with NGINX](#)
[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[WebDAV with NGINX](#)
[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Bandwidth management with NGINX](#)
[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Connection limiting with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Header modification with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[Caching static content](#)

[Removing server name and version](#)

[Extra debug headers](#)

[See also](#)

10. Docker Containers

[Introduction](#)

[Installing Docker](#)

[NGINX web server via Docker](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[NGINX reverse proxy via Docker](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Docker Compose with NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[NGINX load balancing with Docker](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

11. Performance Tuning

[Introduction](#)

[Gzipping content in NGINX](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Enhancing NGINX with keep alive](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Tuning worker processes and connections](#)

[Getting ready](#)

[How to do it...](#)

[Worker processes](#)

[Worker connections](#)

[There's more...](#)

[See also](#)

[Fine tuning basic Linux system limits](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

[Integrating ngx_pagespeed](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

12. [OpenResty](#)

[Introduction](#)

[Installing OpenResty](#)

[Getting ready](#)

[How to do it...](#)

[CentOS](#)

[Ubuntu](#)

[How it works...](#)

[See also](#)

[Getting started with OpenResty Lua](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Lua microservices with OpenResty](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Simple hit counter with a Redis backend](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Powering API Gateways with OpenResty](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

13. NGINX Plus – The Commercial Offering

[Introduction](#)

[Installing NGINX Plus](#)

[Getting ready](#)

How to do it...
CentOS

Ubuntu

See also

Real-time server activity monitoring
Getting ready

How to do it...

How it works...

There's more...

See also

Dynamic config reloading
Getting ready

How to do it...

How it works...

There's more...

See also

Session persistence
Getting ready

How to do it...
Cookie-based tracking

Learn-based tracking

Route-based tracking

How it works...

See also

Preface

NGINX Cookbook covers the basics of configuring NGINX as a web server for use with common web frameworks such as WordPress and Ruby on Rails, through to utilization as a reverse proxy. Designed as a go-to reference guide, this book will give you practical answers based on real-world deployments to get you up and running quickly.

Recipes have also been provided for multiple SSL configurations, different logging scenarios, practical rewrites, and multiple load balancing scenarios. Advanced topics include covering bandwidth management, Docker container usage, performance tuning, OpenResty, and the NGINX Plus commercial features.

By the time you've read this book, you will be able to adapt and use a wide variety of NGINX implementations to solve any problems you have.

What this book covers

[Chapter 1](#), *Let's Get Started*, goes through some of the basics of NGINX as a refresher. It's aimed as an entry point so that there's no assumed knowledge when we move onto some of the more complex structures.

[Chapter 2](#), *Common PHP Scenarios*, covers examples of the more common PHP scenarios and how to implement them with NGINX. The readers will learn how to configure NGINX and how to deploy a basic site.

[Chapter 3](#), *Common Frameworks*, covers non-PHP-based frameworks. It will help the readers to understand and implement all of the common non-PHP-based platforms via NGINX.

[Chapter 4](#), *All About SSLs*, covers installing the various SSL certificates via NGINX and also covers the configuration required to tweak it for certain scenarios.

[Chapter 5](#), *Logging*, explains that monitoring for errors and access patterns are fundamental to running a server.

[Chapter 6](#), *Rewrites*, covers how rewrites work and also specific implementations of many of the common scenarios. It will be full of specific, practical examples based on real-world scenarios.

[Chapter 7](#), *Reverse Proxy*, covers a basic proxy with specific examples of caching and content expiry. This chapter will explain how to configure NGINX as a reverse proxy, content caching, monitoring cache status, microcaching, and many more important scenarios.

[Chapter 8](#), *Load Balancing*, talks about the load balancing components of NGINX and how to implement them for specific scenarios. In this chapter, you will learn the three important load balancing techniques—round-robin, least connection, and hash-based load balancing.

[Chapter 9](#), *Advanced Features*, covers some of the lesser used features of NGINX,

why they're available, and then how to implement them. This chapter includes authentication, WebDAV, bandwidth management, connection limiting, and header modification with NGINX.

[Chapter 10](#), *Docker Containers*, runs you through real-world scenarios of using NGINX within a container. It will provide basic Dockerfile configs for common scenarios.

[Chapter 11](#), *Performance Tuning*, is designed to build upon the existing NGINX configurations and enable specific performance enhancements.

[Chapter 12](#), *OpenResty*, introduces the concept of OpenResty, a combination of NGINX, Lua scripting, and several additional third-party modules all packaged up ready to use.

[Chapter 13](#), *NGINX Plus – The Commercial Offering*, shows the readers what features are in the Plus version, as well as how to implement them.

What you need for this book

The following is the list of software you require to go through the recipes covered in this book:

- NGINX
- PHP 7
- Ubuntu/CentOS/RHEL

Who this book is for

This book is aimed at beginner-to-medium developers who are just getting started with NGINX. It assumes that they already understand the basics of how a web server works and how basic networking works.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it..., How it works..., There's more..., and See also). To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To install the latest NGINX release, add the NGINX mainline repository by adding the following to

```
/etc/yum.repos.d/nginx.repo."
```

A block of code is set as follows: server {
listen 80;
server_name server.yourdomain.com;
access_log /var/log/nginx/log/host.access.log combined;

```
location / {  
root /var/www/html;  
index index.html;  
}  
}
```

Any command-line input or output is written as follows: **mkdir -p
/var/www/vhosts**

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "As our configuration is very simple, we can simply accept the default settings and hit Create."



Warnings or important notes appear 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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply email feedback@packtpub.com, and mention the book's title in 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 at 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.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the book's web page at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account. Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/NGINX-Cookbook>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/NGINXCookbook_ColorImages.pdf.

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 could 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/submit-errata>, 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 to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Let's Get Started

In this chapter, we will cover the following recipes:

- A quick installation guide
- Configuring NGINX
- Stopping/starting NGINX
- Enabling modules
- Deploying a basic website
- Basic monitoring
- Real-time statistics

Introduction

NGINX is a powerful software suite which has progressed well beyond a basic web server package. Some of the additional features, such as the reverse proxy and load balancing options, are well known.

Originally designed to tackle the C10k problem of handling 10,000 concurrent connections, NGINX differentiated itself from Apache with an event-driven architecture. While Apache 2.4 added event-driven processing also, there are a number of distinct differences where NGINX still remains more flexible.

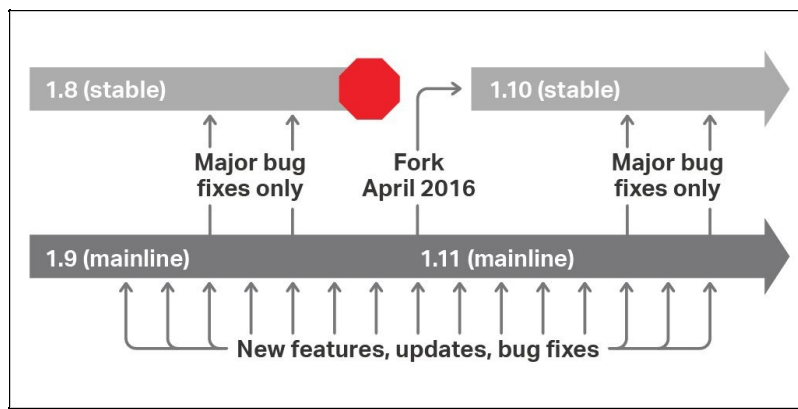
This book describes how to use NGINX in a number of different scenarios and is aimed at providing you with a working solution rather than being an in-depth review of all NGINX features. If you're unfamiliar with NGINX, I highly recommend that you read *Nginx HTTP Server - Third Edition*, by Clément Nedelcu, also published by *Packt Publishing*.



You can also read the official documentation here:
<http://nginx.org/en/docs/>

A quick installation guide

Since the mainline release (currently 1.11.19) has all of the latest features, you'll need to install it directly from the NGINX repositories. Thankfully, NGINX is kind enough to provide **Red Hat Enterprise Linux (RHEL)**, CentOS, **SUSE Linux Enterprise Server (SLES)**, Debian, and Ubuntu repositories, as well as OS X and Windows binaries.



Mainline versus stable

(source: <https://nginx-com-uploads.s3.amazonaws.com/wp-content/uploads/2016/04/NGINX-1.11-mainline-1.10-stable.png>)

The stable and mainline branches don't necessarily reflect system stability, but configuration and module integration stability. Unless you have third-party integration which requires the stable release, we highly recommend the mainline release.

How to do it...

Different Linux distributions have varying package managers, so we'll briefly cover the installation procedures for the more commonly used ones. If the distribution you use isn't covered here, refer to the official NGINX documentation for further guidance.

Packages – RHEL/CentOS

To install the latest NGINX release, add the NGINX mainline repository by adding the following to `/etc/yum.repos.d/nginx.repo`: `[nginx] name=nginx repo baseurl=http://nginx.org/packages/mainline/OS/OSRELEASE/$basearch/ gpgcheck=0 enabled=1`

You'll also need to replace `os` with either `rhel` or `centos`, and replace `OSRELEASE` with `5`, `6`, or `7`, for your correct release.



You can check your version by running `cat /etc/redhat-release`.

Once you have the repository installed, refresh the packages and then install NGINX.

```
| yum update  
| yum install nginx
```

If you have any issues, double check your repository for the correct syntax.



For further information, refer to the official documentation at http://nginx.org/en/linux_packages.html#mainline.

Packages – Debian/Ubuntu

First, download the NGINX signing key for the packages and install it:

```
| wget http://nginx.org/keys/nginx_signing.key  
| apt-key add nginx_signing.key
```

Then, using your preferred Linux editor, we can add the sources to

`/etc/apt/sources.list.d/nginx.list`:

```
| deb http://nginx.org/packages/mainline/debian/ codename nginx  
| deb-src http://nginx.org/packages/mainline/debian/ codename nginx
```

Replace `codename` with the release name; for example, if you're using Debian 8, this will be set to `jessie`.

For Ubuntu-based systems, you'll need to use the following:

```
| deb http://nginx.org/packages/mainline/ubuntu/ codename nginx  
| deb-src http://nginx.org/packages/mainline/ubuntu/ codename nginx
```

Replace `codename` with the release name; for example, if you're using Ubuntu 14.04, this will be set to `trusty`.

After adding the new source, we can then update the `apt` database and install NGINX:

```
| apt-get update  
| apt-get install nginx
```

Installation should now be complete.

Compiling from scratch

Although having the precompiled packages is nice, not all of the modules are available out of the box. NGINX requires you to compile these into the NGINX installation and it's not a simple module like Apache.

You can simply build from source without any of the packaging tools for CentOS or Debian, however, it makes upgrades and compatibility more difficult. By default, user compiled programs will default to `/usr/local`, which means that any documentation which refers to the package defaults (`/usr/etc`) will be incorrect.

My preference is to base the build on the official package sources, rather than the plain source code. There aren't many extra steps involved, but it makes the ongoing management much easier. If you're looking for vanilla build instructions (without packages), these are easily available on the web.



These examples require you to have the mainline repositories already installed.

Debian/Ubuntu

On Ubuntu/Debian, install the required build tools:

```
| apt-get install devscripts
```

This will install quite a few packages on your system, so if you're trying to keep your production environment lean, then I'd recommend that you use a separate build box to complete this.

We can now install the build prerequisites for NGINX: **apt-get build-dep nginx**

Once you have the required build dependencies, we can now get a copy of the source code. Again, rather than the plain TAR file, we're going to get the packaged variant so that we can easily build them. Here's how we do it: **mkdir**

```
~/nginxbuild
```

```
cd ~/nginxbuild
```

```
apt-get source nginx
```

You should now have a directory with the original TAR file, the Debian description, and any Debian specific patches. The `apt-get source` command will automatically extract and apply patches, as required, into a source directory.

To build without any changes, enter the directory and create the packages: **cd**
nginx-1.9.10/
fakeroot debian/rules binary

Compiling the code may take a while, depending on how many processors your workstation or server has. Once it has compiled, you should see two binaries in the parent (`nginxbuild`) directory. The resulting files should be:

- `nginx-dbg_1.9.10-1~jessie_amd64.deb`
- `nginx_1.9.10-1~jessie_amd64.deb`

You can now install NGINX via the newly compiled package: **sudo dpkg -i**
nginx_1.9.10-1~jessie_amd64.deb

CentOS/RHEL

Like the Debian build process, first we'll need to install the package build tools and the additional **Extra Packages For Enterprise Linux (EPEL)** repository:

```
| sudo yum install yum-utils epel-release mock
```

Next, update `/etc/yum.repos.d/nginx.repo` and add the additional source repository:

```
| [nginx-source]
| name=nginx source repo
| baseurl=http://nginx.org/packages/mainline/centos/7/SRPMS/
| gpgcheck=0
| enabled=1
```

In this example, we'll be using a CentOS 7-based release. Refer to the *Packages – RHEL/CentOS* section for how to modify it for other CentOS versions.

With the updated repository, we then create a directory for the build, and download the **Source RPM (SRPM)**:

```
| mkdir ~/nginxbuild
| cd ~/nginxbuild
| yumdownloader --source nginx
```

Next, download the required packages to complete the build:

```
| yum-builddep nginx
```

Once all of the development packages have been downloaded, we can now extract the files from the SRPM:

```
| rpm2cpio nginx-1.9.10-1.el7.ngx.src.rpm | cpio -idmv
```



Note that the name of your directory may vary based on the version of NGINX you have installed. For instance, here it is `nginx-1.9.10` as I have installed NGINX 1.9.10.

You should see an output of the source files similar to this:

```
[root@nginxbuildvm nginxbuild]# rpm2cpio nginx-1.9.10-1.el7.ngx.src.rpm | cpio -idmv
COPYRIGHT
logrotate
nginx-1.9.10.tar.gz
nginx-debug.service
nginx-debug.sysconf
nginx.conf
nginx.init.in
nginx.service
nginx.spec
nginx.suse.logrotate
nginx.sysconf
nginx.upgrade.sh
nginx.vh.default.conf
1790 blocks
[root@nginxbuildvm nginxbuild]#
```

If we want to update the configuration and apply a patch or change one of the defaults, then this can simply be done by editing the files.

We can now rebuild these files from source using `mock`, which is a tool for building packages. The advantage of `mock` is that all of the development dependencies are contained within a *chrooted* environment, so it doesn't clutter your main installation. This *chrooted* environment can be cleaned and removed without any impact on the host system, which is great if you want repeatable builds.

To build, we run the following command:

```
| mock --buildsrpm --spec ~/nginxbuild/nginx.spec --sources ~/nginxbuild
```

This generates the SRPMs, and they will be located in the `/var/lib/mock/epel-7-x86_64/result` directory, along with the associated log files. Now that we have a rebuilt SRPM, we can now compile it. Again, we're going to use `mock` so that everything is neatly contained:

```
| mock --no-clean --rebuild var/lib/mock/epel-7-x86_64/result/nginx-1.9.11-1.el7.ngx.src.
```

Depending on your processing power, this may take five minutes or more to complete. Once the build is complete, you should see the resultant binary RPM as well as a debug RPM in the `/var/lib/mock/epel-7-x86_64` directory. Here's an example:

```
-rw-rw-r-- 1 demo mock 159K Feb 10 20:59 build.log
-rw-r--r-- 1 demo mock 889K Feb 10 20:57 nginx-1.9.11-1.el7.ngx.src.rpm
-rw-r--r-- 1 demo mock 803K Feb 10 20:59 nginx-1.9.11-1.el7.ngx.x86_64.rpm
-rw-r--r-- 1 demo mock 3.1M Feb 10 20:59 nginx-debuginfo-1.9.11-1.el7.ngx.x86_64.rpm
```

```
| -rw-rw-r-- 1 demo mock 45K Feb 10 20:59 root.log  
| -rw-rw-r-- 1 demo mock 1000 Feb 10 20:59 state.log
```

Now that we have the new binary file, we can install it via `yum`:

```
| sudo yum install /var/lib/mock/epel-7-x86_64/result/nginx-1.9.11-1.ngx.x86_64.rpm
```



It's preferable to use `yum` over `rpm` to install the packages, as it can also install any dependencies.

You should now have a fully installed NGINX installation, which you compiled from source.

Testing

Regardless of your installation method, once you have NGINX up and running, you should be able to browse to it via the IP address and/or **Fully Qualified Domain Name (FQDN)** and see something very similar to what is shown here:



Default NGINX page

To start, stop, and restart NGINX (if installed using official binaries), you can use the standard Linux init systems. There's a very slight variance between the different OS versions, so it's important to ensure you're using the correct command for the correct variant.



As Ubuntu switched to `systemd` as the default init system from 15.04, make sure you double check the version you're using.

How to do it...

Here's a quick reference table of the available commands:

Activity/OS	CentOS / RedHat 6	CentOS / RedHat 7	Ubuntu 14.04 / Debian 8
Start NGINX	<code>service nginx start</code>	<code>systemctl start nginx</code>	<code>service nginx start</code>
Stop NGINX	<code>service nginx stop</code>	<code>systemctl stop nginx</code>	<code>service nginx stop</code>
Restart NGINX	<code>service nginx restart</code>	<code>systemctl restart nginx</code>	<code>service nginx restart</code>
Reload NGINX	<code>service nginx reload</code>	N/A	N/A

Some modifications to NGINX will require a full restart, whereas others only need the configuration reloaded. In most instances where a configuration file has been modified, a reload is all which is required. NGINX will fork a new set of worker processes, allowing existing workers to complete and cleanly exit so that there is no downtime.

There's more...

We can check the NGINX configuration files after changes are made to ensure the syntax is correct. To do this, we run the following: **/usr/sbin/nginx -t**

You should see the following if everything is correct:

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

If you have any errors, double check your configuration for syntax errors on the lines indicated.

Configuring NGINX

Editing the configuration for NGINX is paramount to the way it operates. To integrate NGINX with your particular website or need, you'll need to edit a number of areas within the configuration files. To get started, we'll go through the basics here.

```
user nginx;
```

```
worker_processes 1;
```

```
error_log /var/log/nginx/error.log warn; pid /var/run/nginx.pid;
```

```
events {
```

```
    worker_connections 1024; }
```

```
http {
```

```
    include /etc/nginx/mime.types; default_type application/octet-stream;
```

```
    log_format main '$remote_addr - $remote_user [$time_local] <br/> "$request" '
```

```
    '$status $body_bytes_sent "$http_referer" '
```

```
    '"$http_user_agent" "$http_x_forwarded_for";
```

```
    access_log /var/log/nginx/access.log main;
```

```
    sendfile on; #tcp_nopush on;
```

```
    keepalive_timeout 65;
```

```
    #gzip on;
```

```
include /etc/nginx/conf.d/*.conf; }
```

The configuration files have two main components in them—**simple directives** and **block directives**. Simple directives are one-line items which are simple name and value, followed by a semicolon (;). A block directive has a set of brackets and allows configuration items to be set within a specific context. This makes the configuration files easier to follow, especially as they get more complex.

How it works...

Here are a few of the key configuration items. Firstly, `user nginx` defines the user in which NGINX will run as. This is important to note if you have a server-side script which requires the ability to write files and a user will also require permission to read the files.

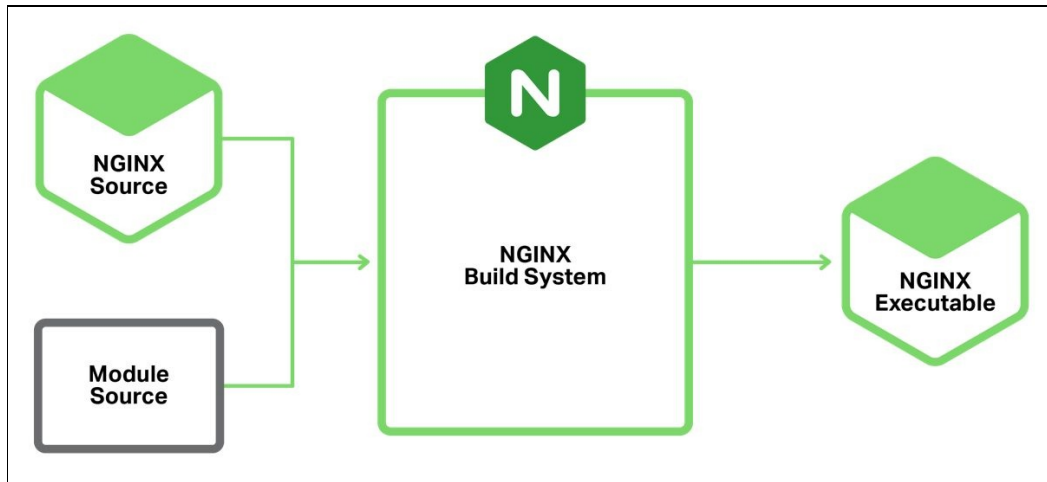
Secondly, `worker_processes` sets the number of worker processes that NGINX will start. While a default of `1` doesn't sound very high, the event-driven nature means that this certainly won't be a limitation initially. The optimal number of processes depends on many factors, but an easy starting reference is to go by the number of CPU cores your server has.

Next, `worker_connections` is the maximum amount of simultaneous connections that a worker process can open. In the default configuration, this is set to `1024` concurrent connections.

Lastly, the `include /etc/nginx/conf.d/*.conf;` line tells NGINX to load all of the `.conf` files as if they were all part of the main `nginx.conf` file. This allows you to separate the configuration for different sites.

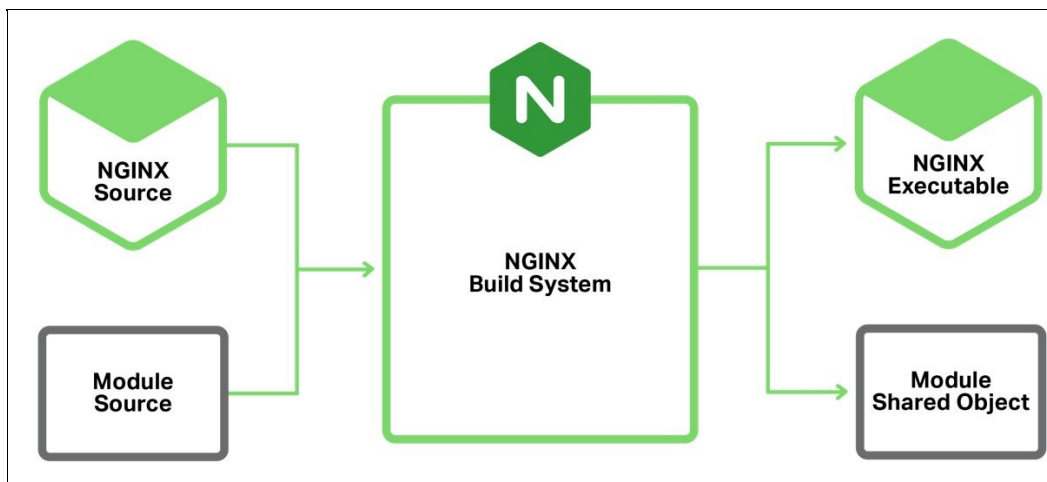
Enabling modules

By default, not every module for NGINX has been compiled and is available. As of version 1.9.11 (released in February 2016), NGINX added dynamic modules, similar to the **Dynamic Shared Objects (DSO)** like the Apache HTTP server.



Previous NGINX build process

Previous to this, you needed to recompile NGINX every time you updated a module, so this is a big step forward. Being statically compiled into the main binary also meant that each instance of NGINX loaded all the modules internally, regardless of whether you needed them or not. This is why the mail modules were never compiled in with the standard binaries.



NGINX new dynamic modules

How to do it...

However, even though 1.9.11 added the dynamic modules, none of the standard modules are dynamic by default. To make them into dynamic modules, you'll need to update the configure options. For example, to make the mail module dynamic, update `--with-mail` to `--with-mail=dynamic`.

The main `nginx.conf` configuration file will then need the following:

```
| load_module "modules/ngx_mail_module.so";
```



Official Link: <https://www.nginx.com/resources/wiki/extending/>

See also

There are a number of third-party modules for NGINX, some of which we'll be using later in this book and can be found at <https://www.nginx.com/resources/wiki/modules/>.

Deploying a basic site

If you have a static website, this is very easy to deploy with NGINX. With systems such as Jekyll (which powers the GitHub Pages), static site deployments can be easy to generate and are far less hassle when it comes to security and exploits. Quite simply, a static website can't be hacked and doesn't suffer from any performance issues.

```
<strong> mkdir -p /var/www/vhosts</strong>
```

```
<strong> chmod -R o+r /var/www/vhosts<br/> </strong>
```

```
<strong> chown -R nginx:nginx /var/www/vhosts</strong>
```

5. From your web browser, browse the site and check that it's working.

How it works...

Let's go through this setup file to understand each directive:

- `listen 80;`: This defines the port which NGINX will listen to. Port 80 is the default standard for HTTP, which is why it doesn't need to be specified in the browser URL.
- `server_name server.yourname.com;`: This directive tells the server what hostname to match from the request. This allows you to run name-based virtual servers from one IP address, but with different domain names. You can also use different aliases here; for example, you can have both `www.yourname.com` and `yourname.com`.
- `access_log /var/log/nginx/log/host.access.log combined;`: The access log records all client access to the site, stores it in the specified file (the second parameter), and uses the third parameter to define the format of the log (`combined` is the default).
- `location`: Lastly, we have a `location` block directive. This one is for a root directive (represented by `/`), meaning everything in the URL path. There are then two directives contained within this block—the first is the `root` directive. This defines where NGINX should look for the files.
- `index`: The second is the `index` directive. This lets NGINX know what name of a file to try if it hasn't been specified in the path. For example, if you put `http://server.yourname.com/` into your browser, NGINX will try to load `http://server.yourname.com/index.html` instead of displaying a 404 error.

Basic monitoring

Monitoring both the performance and uptime of a web server is paramount when you want to ensure consistent performance. There are a number of ways both these aspects can be monitored, all with varying levels of complexity and information. We'll focus on some of the simpler examples to give you a starting point to go forward with.

How to do it...

We can enable the basic NGINX `stub_status` page to give some rudimentary statistics and service status. To enable, edit your site config and add the following: `location = /nginx_status { stub_status on; access_log off; allow <YOURIPADDRESS>; deny all; }`

To prevent information leakage about your system, we have added the `allow` command. This should be your IP address. This is followed by the `deny all` command to prevent anyone else from loading the URL. We've also turned off access logs for this URL to save space.

After reloading your configuration (hint: `systemctl reload nginx` for systemd-based OS), you can now load the new URL `/nginx_status` in your browser.

You should see something like the following:

```
Active connections: 6
server accepts handled requests
 81 81 177
Reading: 0 Writing: 1 Waiting: 5
```

How it works...

Let's take apart the details line-by-line:

- The `Active connections` line lists the amount of connections to the server. For a quiet server, this could be less than a dozen. For a busy server, expect this to be in the hundreds.
- The `server accepts handled requests` line is little confusing, since it's represented by three numbers (`81`, `81`, and `177` in this example). The first number represents the amount of accepted connections. The second number represents the total number of handled connections. Unless there's a resource limitation, the number of accepted and handled connections should be the same. Next, we have the total number of client requests.
- The last line represents the state of the active connections. `Reading` means NGINX is reading the request headers, `Writing` means NGINX is writing data back to the client, and `Waiting` means that the client is now idle but still has the connection open (due to keep-alives).

Real-time statistics

When your web server is important to you, you'll want to see what's going on in real-time. One great utility to do this is `ngxtop`. This program monitors your real-time access log in to display useful metrics such as the number of requests per second, HTTP statuses served, and pages/URLs served. This information can be a great aid to determine what your top pages are and if there is an increased rate of errors.

How to do it...

To install `ngxtop`, you'll first need to install the Python package manager, `pip`. On a Debian/Ubuntu-based system, you'll need to run the following: **`apt-get install python-pip`**

For a Red Hat- / CentOS-based system, the EPEL repository is required first. To install, run the following:

```
| yum install epel-release  
| yum install python-pip
```

Once you have `pip` installed, you can now install `ngxtop`:

```
| pip install ngxtop
```

This will download the required files and install `ngxtop` for you. It's now ready to use.

How it works...

The `ngxtop` utility can be simply called from the command line, and it will attempt to read the log file location from the configuration. However, if you're running virtual hosts, it may not read the access log location correctly. The easiest way in this scenario is to manually specify the log location.

Consider the following example:

```
| ngxtop -l /var/log/nginx/access.log
```

This will display a console-based view of what URLs are being accessed. Here's a basic example:

```
running for 132 seconds, 34 records processed: 0.26 req/sec
```

Summary:						
count	avg_bytes_sent	2xx	3xx	4xx	5xx	
34	6245.529	25	1	8	0	

Detailed:						
request_path	count	avg_bytes_sent	2xx	3xx	4xx	5xx
/	32	6370.406	24	0	8	0
/test	1	8490.000	1	0	0	0
/test/	1	5.000	0	1	0	0

This will refresh every second, giving you near instantaneous information about what URLs NGINX is serving, but that's not all. The `ngxtop` utility is quite configurable and there are a number of different ways it can display information.

We can filter only the `404` pages with the following:

```
| ngxtop -l /var/log/nginx/access.log --filter 'status == 404'
```

Only those URLs which had a `404` are now going to be displayed within `ngxtop`. Here's an example output:

```
running for 28 seconds, 3 records processed: 0.11 req/sec

Summary:
| count | avg_bytes_sent | 2xx | 3xx | 4xx | 5xx |
|-----+-----+-----+-----+-----+-----|
|      3 |      17525.333 |    0 |    0 |    3 |    0 |

Detailed:
| request_path | count | avg_bytes_sent | 2xx | 3xx | 4xx | 5xx |
|-----+-----+-----+-----+-----+-----+-----|
| /asdfasdfasdf |      1 |      17525.000 |    0 |    0 |    1 |    0 |
| /doesnt_exist  |      1 |      17525.000 |    0 |    0 |    1 |    0 |
| /nor_does_this |      1 |      17526.000 |    0 |    0 |    1 |    0 |
```

There are quite a number of other options easy to tailor to your scenario, especially if you know what you're hunting for.

See also

To know more about `ngxtop`, refer to <https://github.com/lebinh/nginx-top>.

Common PHP Scenarios

In this chapter, we will cover the following recipes:

- Configuring NGINX for WordPress
- WordPress multisite with NGINX
- Running Drupal using NGINX
- Using NGINX with MediaWiki
- Using Magento with NGINX
- Configuring NGINX for Joomla

Introduction

PHP is a thoroughly tested product to use with NGINX because it is the most popular web-based programming language. It powers sites, such as Facebook, Wikipedia, and every WordPress-based site, and its popularity hasn't faded as other languages have grown.

In this chapter, we'll go through examples of the more common PHP scenarios and how to implement them with NGINX. As WordPress is the most popular of the PHP systems, I've included some additional information to help with troubleshooting. Even if you're not using WordPress, some of this information may be helpful if you run into issues with other PHP frameworks.

Most of the recipes expect that you have a working understanding of PHP systems, so not all of the setup steps for the systems will be covered. This is to keep the book succinct and allow the focus to be on the NGINX components.

In order to keep the configurations as simple as possible, I haven't included details such as cache headers or SSL configurations in these recipes. This will be covered in later chapters and the full working configurations will be made available via <https://github.com/timbutler/nginxcookbook/>.

Configuring NGINX for WordPress

Covering nearly 30 percent of all websites, WordPress is certainly the **Content Management System (CMS)** of choice by many. Although it came from a blogging background, WordPress is a very powerful CMS for all content types and powers some of the world's busiest websites.

By combining it with NGINX, you can deploy a highly scalable web platform.



You can view the official WordPress documentation on NGINX at <https://codex.wordpress.org/Nginx>.

We'll also cover some of the more complex WordPress scenarios, including multisite configurations with subdomains and directories.

Let's get started.

Getting ready

To compile PHP code and run it via NGINX, the preferred method is via **PHP-FPM**, a high-speed **FastCGI Process Manager**. We'll also need to install PHP itself and, for the sake of simplicity, we'll stick with the OS-supplied version. Those seeking the highest possible performance should ensure they're running PHP 7 (released December 3, 2015), which can offer a 2-3 times speed improvement for WordPress, while at the same time being up to four times more memory efficient compared to PHP 5.6.

To install PHP-FPM, you should run the following on a Debian/Ubuntu system:
apt-get install php7.0-fpm

For those running CentOS/RHEL, you should run the following:

```
| sudo yum install php-fpm
```

As PHP itself is a prerequisite for the `php-fpm` packages, it will also be installed.



Note: Other packages such as MySQL will be required if you're intending to run this on a single VPS instance. Consult the WordPress documentation for a full list of requirements.

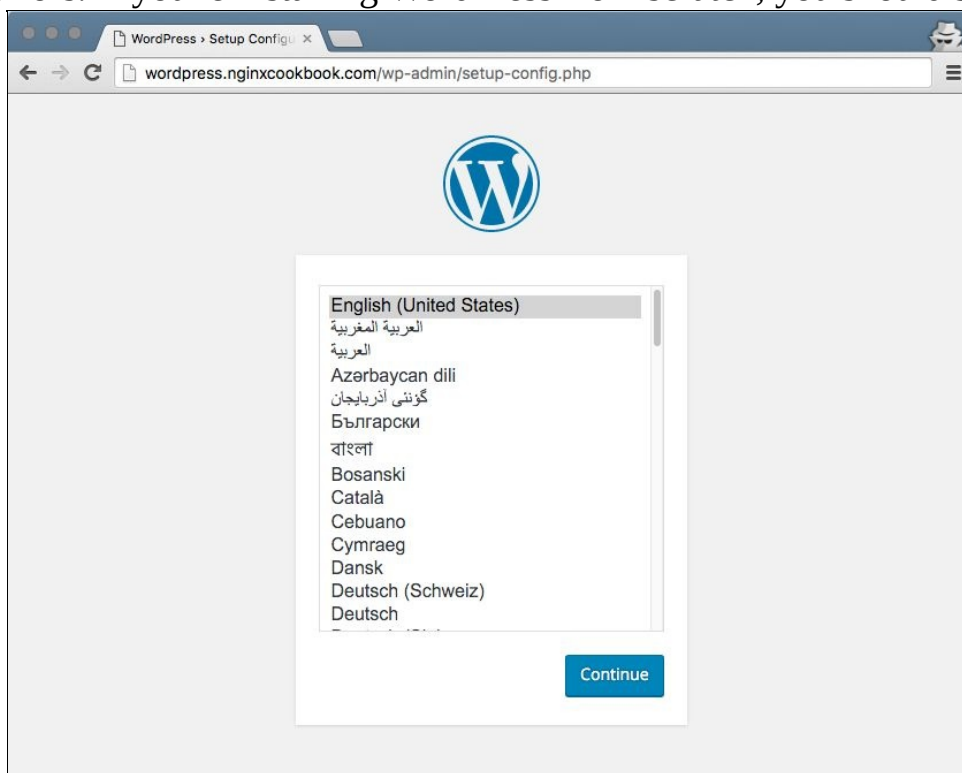
How to do it...

In this instance, we're simply using a standalone WordPress site, which would be deployed in many personal and business scenarios. This is the typical deployment for WordPress.

For ease of management, I've created a dedicated config file just for the WordPress site (`/etc/nginx/conf.d/wordpress.conf`):

```
server { listen 80; server_name wordpressdemo.nginxcookbook.com; access_log /var/log/nginx/access.log combined; location / { root /var/www/html; try_files $uri $uri/ /index.php?$args; } location ~ \.php$ { fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index index.php; fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name; include fastcgi_params; } }
```

Reload NGINX to read the new configuration file and check your log files if there are any errors. If you're installing WordPress from scratch, you should see



the following:

You can complete the WordPress installation if you haven't already.

How it works...

For the most part, the configuration is the same as the static website configuration in [Chapter 1](#), *Let's Get Started*. For the root URL call, we have a new `try_files` directive, which will attempt to load the files in the order specified, but will fall back to the last parameter if they all fail.

For this WordPress example, it means that any static files will be served if they exist on the system, then fall back to `/index.php?args` if this fails.

The `args` rewrite allows the permalinks of the site to be in a much more human form. For example, if you have a working WordPress installation, you can see links such as the one shown in the following screenshot:



Lastly, we process all PHP files via the FastCGI interface to PHP-FPM. In the preceding example, we're referencing the Ubuntu/Debian standard; if you're running CentOS/RHEL, then the path will be `/var/run/php-fpm.sock`.

NGINX is simply *proxying* the connection to the PHP-FPM instance, rather than being part of NGINX itself. This separation allows for greater resource control, especially since the number of incoming requests to the web server doesn't necessarily match the number of PHP requests for a typical website.

There's more...

Take care when copying and pasting any configuration files. It's very easy to miss something and have one thing slightly different in your environment, which will cause issues with the website working as expected. Here's a quick lookup table of various other issues which you may come across:

Error	What to check
502 Bad Gateway	File ownership permissions for the PHP-FPM socket file
404 File Not Found	Check for the missing <code>index index.php</code> directive
403 Forbidden	Check for the correct path in the <code>root</code> directive and/or check for correct file permissions

Your error log (defaults to `/var/log/nginx/error.log`) will generally contain a lot more detail regarding the issue you're seeing compared with what's displayed in the browser. Make sure you check the error log if you receive any errors.



Hint: NGINX does not support `.htaccess` files. If you see examples on the web referencing a `.htaccess` files, these are Apache specific. Make sure any configurations you're looking at are for NGINX.

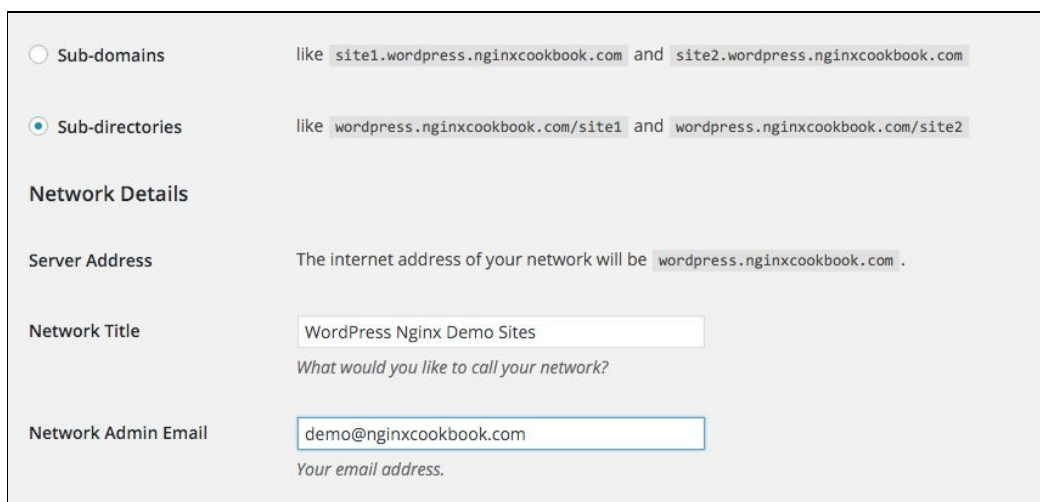
WordPress multisite with NGINX

WordPress **multisites** (also referred to as network sites) allow you to run multiple websites from the one codebase. This can reduce the management burden of having separate WordPress installations when you have similar sites. For example, if you have a sporting site with separate news and staff for different regions, you can use a multisite installation to accomplish this.

How to do it...

To convert a WordPress site into a multisite, you need to add the configuration variable into your config file `wp-config.php`: `define('WP_ALLOW_MULTISITE', true);`

Under the Tools menu, you'll now see an extra menu called Network Setup. This will present you with two main options, Sub-domains and Sub-directories. These are the two different ways the multisite installation will work. The Sub-domains option has the sites separated by domain names, for example, `site1.nginxcookbook.com` and `site2.nginxcookbook.com`. The Sub-directories option means that the sites are separated by directories, for example, `www.nginxcookbook.com/site1` and `www.nginxcookbook.com/site2`.



The screenshot shows the 'Network Setup' screen in WordPress. At the top, there are two radio button options: 'Sub-domains' and 'Sub-directories'. The 'Sub-directories' option is selected. Below these options, there is a section titled 'Network Details'. This section contains three fields: 'Server Address' with a text input containing 'wordpress.nginxcookbook.com', 'Network Title' with a text input containing 'WordPress Nginx Demo Sites', and 'Network Admin Email' with a text input containing 'demo@nginxcookbook.com'. Each field has a small hint text below it: 'The internet address of your network will be', 'What would you like to call your network?', and 'Your email address.' respectively.

There's no functional difference between the two, it's simply an aesthetic choice. However, once you've made your choice, you cannot return to the previous state.

Once you've made the choice, it will then provide the additional code to add to your `wp-config.php` file.

Here's the code for my example, which is subdirectory based:

```
define('MULTISITE', true);
define('SUBDOMAIN_INSTALL', false);
define('DOMAIN_CURRENT_SITE', 'wordpress.nginxcookbook.com');
define('PATH_CURRENT_SITE', '/');
```

```
| define('SITE_ID_CURRENT_SITE', 1);  
| define('BLOG_ID_CURRENT_SITE', 1);
```

Because NGINX doesn't support `.htaccess` files, the second part of the WordPress instructions will *not* work. Instead, we need to modify the NGINX configuration to provide the rewrite rules ourselves.

In the existing `/etc/nginx/conf.d/wordpress.conf` file, you'll need to add the following just after the `location /` directive: `if (!-e $request_filename) { rewrite /wp-admin$ $scheme://$host$uri/ permanent; rewrite ^(/[^\/]*)?(/wp-.*)$2 last; rewrite ^(/[^\/]*)?(/*\.\php) $2 last; }`

Although the `if` statements are normally avoided if possible, in this instance, it will ensure the subdirectory multisite configuration works as expected. If you're expecting a few thousand concurrent users on your site, then it may be worthwhile investigating the static mapping of each site. There are plugins to assist with the map generations for this, but they are still more complex compared to the `if` statement.

Subdomains

If you've selected subdomains, your code to put in `wp-config.php` will look like this:

```
define('MULTISITE', true);
define('SUBDOMAIN_INSTALL', true);
define('DOMAIN_CURRENT_SITE', 'wordpressdemo.nginxcookbook.com');
define('PATH_CURRENT_SITE', '/');
define('SITE_ID_CURRENT_SITE', 1);
define('BLOG_ID_CURRENT_SITE', 1);
```

You'll also need to modify the NGINX config as well to add in the wildcard for the server name:

```
| server_name *.wordpressdemo.nginxcookbook.com wordpressdemo.nginxcookbook.com;
```

You can now add in the additional sites, such as

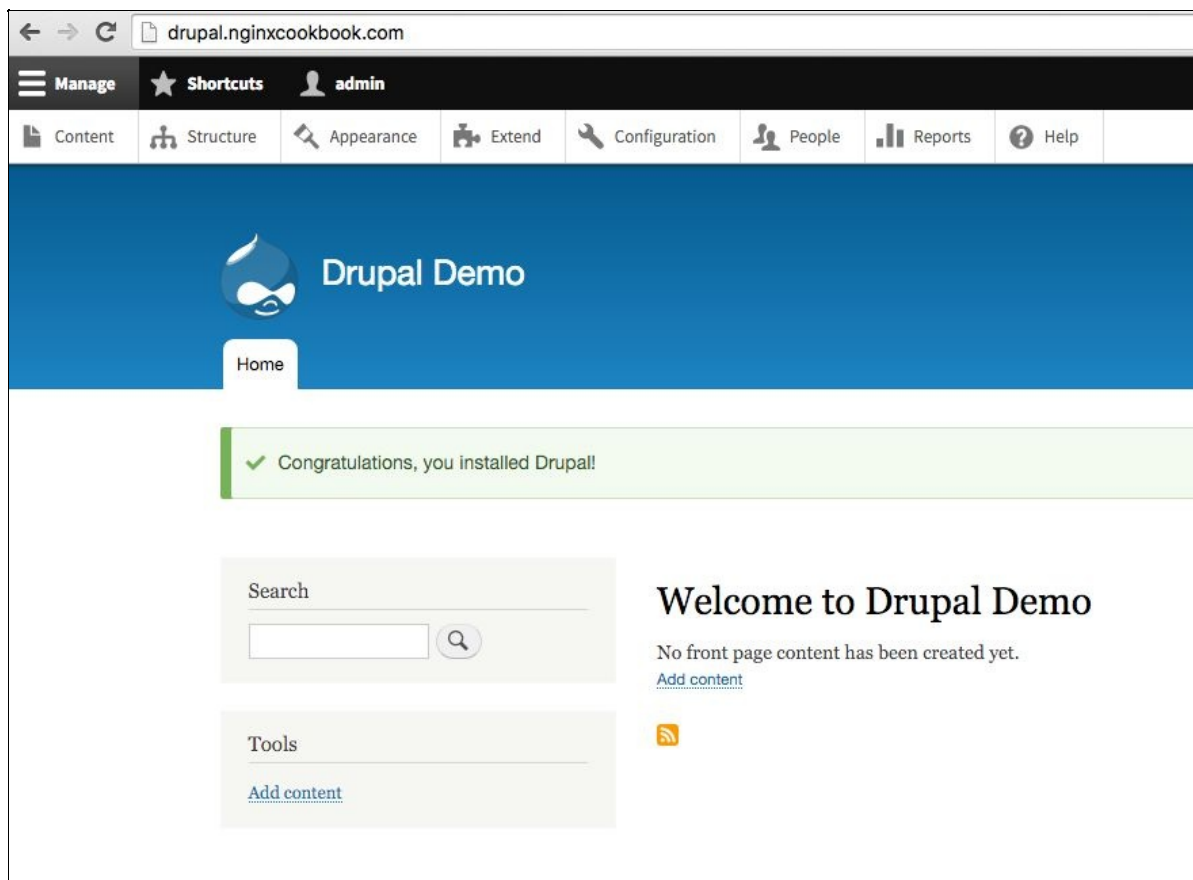
`site1.wordpressdemo.nginxcookbook.com`, and there won't be any changes required for NGINX.

See also

- NGINX recipe page: <https://www.nginx.com/resources/wiki/start/topics/recipes/wordpress/>
- WordPress Codex page: <https://codex.wordpress.org/Nginx>

Running Drupal using NGINX

With version 8 recently released and a community of over 1 million supporters, Drupal remains a popular choice when it comes to a highly flexible and functional CMS platform. Version 8 has over 200 new features compared to version 7, aimed at improving both the usability and manageability of the system. This cookbook will be using version 8.0.5.



Getting ready

This example assumes you already have a working instance of Drupal or are familiar with the installation process. You can also follow the installation guide available at <https://www.drupal.org/documentation/install>.

```
server {  
  
    listen 80; server_name drupal.nginxcookbook.com;  
  
    access_log /var/log/nginx/drupal.access.log combined; index  
index.php;  
  
    root /var/www/html/  
  
    location / {  
  
        try_files $uri $uri/ /index.php?$args; }  
  
    location ~ (^|/)\. {  
  
        return 403; }  
  
    location ~ /vendor/.*\.php$ {  
  
        deny all;  
  
        return 404; }  
  
    location ~ \.php$|^/update.php {  
  
        fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_split_path_info  
^(.+?\.php)(/.*);$ fastcgi_index index.php; fastcgi_param  
SCRIPT_FILENAME <br/> $document_root$fastcgi_script_name;  
include fastcgi_params; }
```

}

How it works...

Based on a simple PHP-FPM structure, we make a few key changes specific to the Drupal environment. The first change is as follows: `location ~ (^|/)\. { return 403; }`

We put a block in for any files beginning with a dot, which are normally hidden and/or system files. This is to prevent accidental information leakage: `location ~ /vendor/.*\.php$ { deny all; return 404; }`

Any PHP file within the `vendor` directory is also blocked, as they shouldn't be called directly. Blocking the PHP files limits any potential exploit opportunity which could be discovered in third-party code.

Lastly, Drupal 8 changed the way the PHP functions are called for updates, which causes any old configuration to break. The `location` directive for the PHP files looks like this: `location ~ \.php$|^/update.php {`

This is to allow the distinct pattern that Drupal uses, where the PHP filename could be midway through the URI.

We also modify how the FastCGI process splits the string, so that we ensure we always get the correct answer:

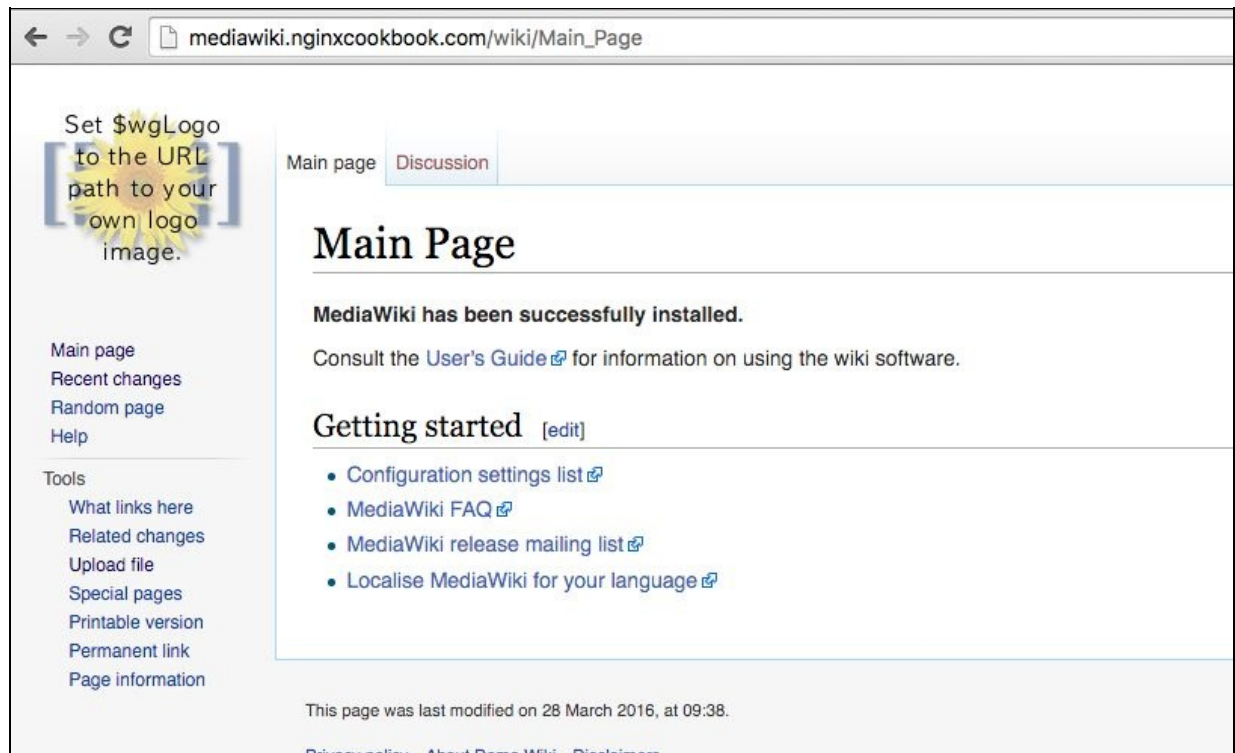
```
| fastcgi_split_path_info ^(.+?\.php)(|.*)$;
```

See also

NGINX recipe: <https://www.nginx.com/resources/wiki/start/topics/recipes/drupal/>

Using NGINX with MediaWiki

MediaWiki, most recognized by its use with Wikipedia, is the most popular open source wiki platform available. With features heavily focused on the ease of editing and sharing content, MediaWiki makes a great system to store information you want to continually edit:



Getting ready

This example assumes you already have a working instance of MediaWiki or are familiar with the installation process. For those unfamiliar with the process, it's available online at https://www.mediawiki.org/wiki/Manual:Installation_guide.

```

server {

    listen 80; server_name mediawiki.nginxcookbook.com;

    access_log /var/log/nginx/mediawiki.access.log combined; index
index.php;

    root /var/www/html;

    location / {

        try_files $uri $uri/ /index.php?$args; }


    location ~ /\.php$ {

        fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index
index.php; fastcgi_param SCRIPT_FILENAME <br/>
$document_root$fastcgi_script_name; include fastcgi_params; }

    }

    $wgArticlePath = "/wiki/$1";

    $wgUsePathInfo = TRUE;

```

This allows the URLs to be rewritten in a much neater format.

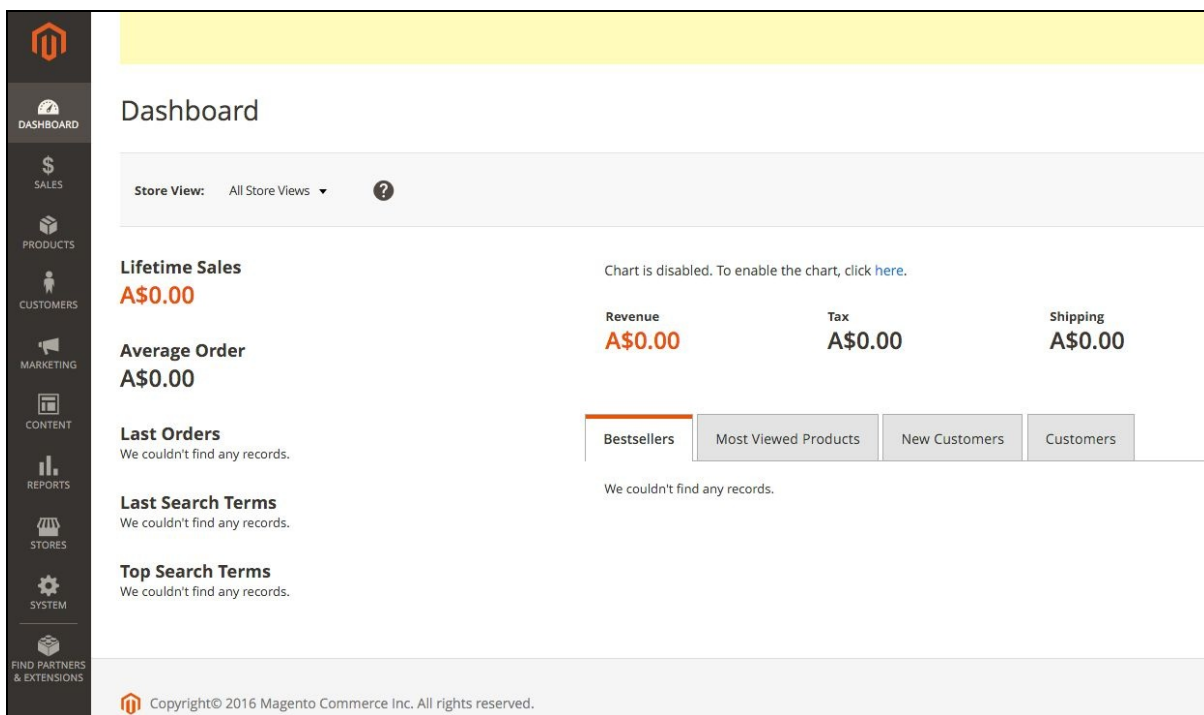
See also

NGINX recipe: <https://www.nginx.com/resources/wiki/start/topics/recipes/mediawiki/>

Using Magento with NGINX

With nearly 30 percent market share, Magento is the most popular e-commerce platform in the world. Due to a number of features and complexity, it's also a very resource-intensive system to use compared to a lightweight alternative. This means that NGINX is an ideal pairing to ensure you have the highest performance possible.

The latest major version of Magento is 2.0, which was nearly a complete rewrite compared to the previous versions. There's still quite a bit of complexity involved too, so make sure that you're ready to take on Magento if you've chosen it for your e-commerce platform:



The screenshot displays the Magento 2.0 Admin Dashboard. On the left is a vertical sidebar with icons and labels for various sections: DASHBOARD, SALES, PRODUCTS, CUSTOMERS, MARKETING, CONTENT, REPORTS, STORES, SYSTEM, and FIND PARTNERS & EXTENSIONS. The main content area is titled 'Dashboard' and features a 'Store View' dropdown set to 'All Store Views'. Below this, the dashboard is divided into several sections. On the left, 'Lifetime Sales' shows 'A\$0.00', 'Average Order' shows 'A\$0.00', 'Last Orders' and 'Last Search Terms' both state 'We couldn't find any records.', and 'Top Search Terms' also states 'We couldn't find any records.'. On the right, a message indicates 'Chart is disabled. To enable the chart, click [here](#).' Below this, three summary cards show 'Revenue A\$0.00', 'Tax A\$0.00', and 'Shipping A\$0.00'. At the bottom right, there are four tabs: 'Bestsellers' (which is active), 'Most Viewed Products', 'New Customers', and 'Customers'. The footer of the dashboard contains the Magento logo and the text 'Copyright© 2016 Magento Commerce Inc. All rights reserved.'

Getting ready

This guide assumes you're familiar with the installation of Magento 2.0 and have a working instance. Although there shouldn't be too many major changes, this recipe has been tested with version 2.0.2.

```
server {  
  
    listen 80;  
  
    server_name magento.nginxcookbook.com; set $MAGE_ROOT  
/var/www/html;  
  
    access_log /var/log/nginx/magento.access.log combined; index  
index.php;  
  
    root $MAGE_ROOT/pub/;  
  
    location / {  
  
        try_files $uri $uri/ /index.php?$args; }  
  
    location ~ ^/(setup|update) {  
  
        root $MAGE_ROOT;  
  
        location ~ ^/(setup|update)/index.php {  
  
            fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index  
index.php; fastcgi_param SCRIPT_FILENAME <br/>  
$document_root$fastcgi_script_name; include fastcgi_params; }  
  
    location ~ ^/(setup|update)/(?!pub/). {  
  
        deny all;  
  
    }  
}
```

```
location ~ ^/(setup|update)/pub/ {
```

```
add_header X-Frame-Options "SAMEORIGIN"; }
```

```
}
```

```
location /static/ {
```

```
expires max;
```

```
if (!-f $request_filename) {
```

```
rewrite ^/static/(version\d*/)?(.*)$ <br/> /static.php?resource=$2  
last; }
```

```
add_header X-Frame-Options "SAMEORIGIN"; }
```

```
location /media/ {
```

```
try_files $uri $uri/ /get.php?$args;
```

```
location ~ ^/media/theme_customization/.*\.xml {
```

```
deny all;
```

```
}
```

```
add_header X-Frame-Options "SAMEORIGIN"; }
```

```
location ~ /\.php$ {
```

```
    fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index  
    index.php; fastcgi_param SCRIPT_FILENAME <br/>  
    $document_root$fastcgi_script_name; include fastcgi_params; }
```

```
}
```

As you can see, this is significantly more complex than a basic WordPress configuration. There are four main sections; the first is to handle the setup and updates, the second is to handle static media (for example, default Magento, CSS, and JavaScript), media files (for example, upload images), and finally how to process PHP files.

How it works...

We'll go through this one in sections. Magento is different from many flat file / single root directory structures, so it requires some slight changes compared to a basic PHP site.

```
| set $MAGE_ROOT /var/www/html;
```

This sets a variable we can easily reference and it means there's only one place we need to update if we move the files:

```
| root    $MAGE_ROOT/pub/;
```

All of the main website files sit in the `pub` subdirectory. This is commonly overlooked when uploading the files to a shared hosting platform such as CPanel or Plesk. Ensure that the main `root` directive points to the `pub` folder, not just the directory of the Magento files. The `root` directive is therefore pointed at the `pub` folder with the preceding configuration line.

Conversely, the setup and update URLs need to have a separate `root` directive to ensure they also point to the correct location:

```
| location (/setup|/upgrade) {  
|     root $MAGE_ROOT;  
| }
```

This sets the `root` back to the default directory, which sits outside the `pub` directory. The easiest way to look at it is to view the setup and upgrade sites as separate websites with their own separate structure. This is why the directive block also has the following:

```
| location ~ ^/(setup|update)/index.php {  
|     fastcgi_pass    unix:/var/run/php7.0-fpm.sock;  
|     fastcgi_index   index.php;  
|     fastcgi_param   SCRIPT_FILENAME  
|         $document_root$fastcgi_script_name;  
|     include         fastcgi_params;  
| }
```

We only allow access to `index.php` within the `setup/update` directories, and then deny access to any *nonpub* file:

```
| location ~ ^/(setup|update)/(?!pub/). {  
    deny all;  
}
```

This will give a 403 error if a malicious user or script attempts to access files outside the `pub` directory.

It also ensures that all requests come from the same frame, which will prevent clickjacking:

```
| add_header X-Frame-Options SAMEORIGIN;
```

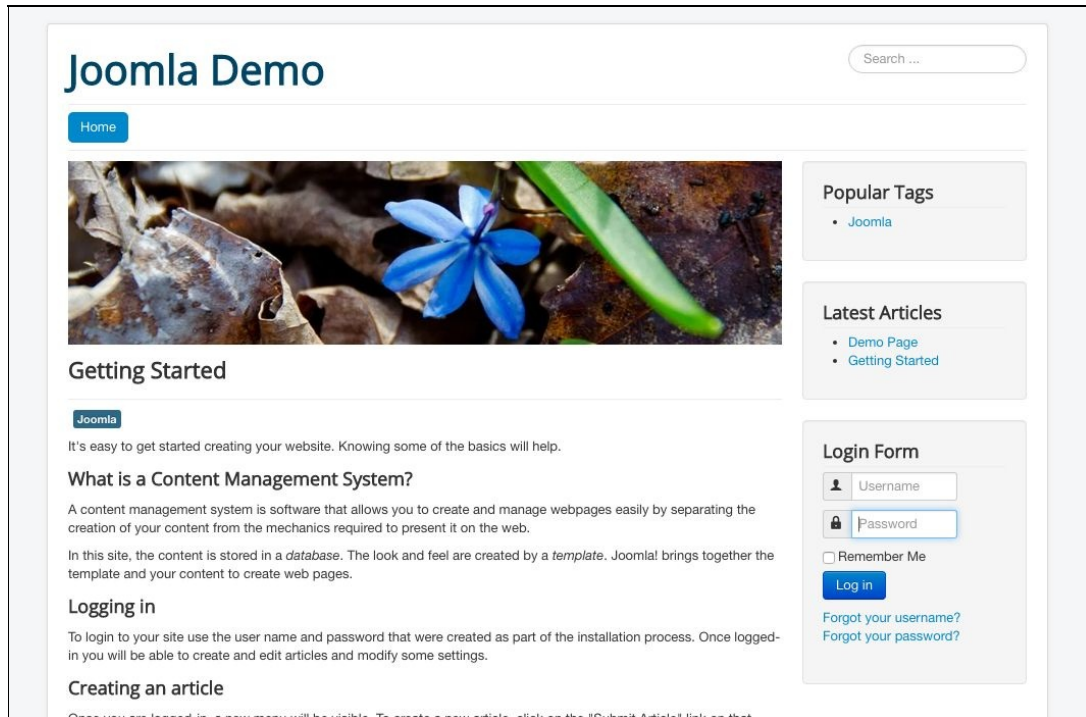
The static and media sections are both fairly similar in how they operate. Headers are set for caching (explained in more detail in [Chapter 7, Reverse Proxy](#)) and the calls are wrapped through a PHP function (`static.php` or `get.php`) to allow Magento to perform tasks such as file merging and minification. It can result in a slightly slower first hit, but as it caches the result each subsequent request should be very fast.

See also

Magento sample configuration: <https://github.com/magento/magento2/blob/develop/nginx.conf.sample>

Configuring NGINX for Joomla

With the recent release of version 3.5, Joomla is still one of the most popular CMS platforms in the world. This latest release features full PHP 7 support, drag-and-drop images, and more:



Getting ready

This assumes you have a working Joomla installation and have located the files at `/var/www/html/`. If you need an installation guide, try the official documentation available at https://docs.joomla.org/J3.x:Installing_Joomla.

```

server {

    listen 80;

    server_name joomla.nginxcookbook.com;

    access_log /var/log/nginx/joomla.access.log combined; index
index.php;

    root /var/www/html/;

    location / {

        try_files $uri $uri/ /index.php?$args; }

    location ~ /\.php$ {

        fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index
index.php; fastcgi_param SCRIPT_FILENAME <br/>
$document_root$fastcgi_script_name; include fastcgi_params; }

    location ^~ /cache/ {

        deny all;

    }

}

```

To enable clean URLs, you also need to ensure that URL rewriting is

enabled:

Search Engine Friendly URLs	<input checked="" type="radio"/> Yes	<input type="radio"/> No
Use URL Rewriting	<input checked="" type="radio"/> Yes	<input type="radio"/> No

See also

Joomla docs: <https://docs.joomla.org/Nginx>

Common Frameworks

In this chapter, we will cover the following recipes:

- Setting up Django with NGINX
- Setting up NGINX with Express
- Running Ruby on Rails with NGINX
- Easy Flask with NGINX
- Laravel via NGINX
- Meteor applications with NGINX
- High speed Beego with NGINX

Introduction

The boom of web-based services and websites beyond just static content has spawned a number of web frameworks to tackle some of these more complex scenarios. Some have started from the newspaper world, where there are very tight deadlines, while others have focused on performance as their reasoning.

In this chapter, we'll go through the configurations of NGINX to work with each of these various frameworks in order to have an easy-to-follow configuration. Like the previous chapter, I won't be going through the steps of setting up the frameworks themselves, but focusing on the NGINX side only.

Let's get started.

Setting up Django with NGINX

Django rose to fame when a small US newspaper firm open sourced their application back in 2005. From there, Django has grown into the most widely used Python-based web framework. Still used by companies such as Instagram and Pinterest, the relevancy of Django is still as strong as ever.

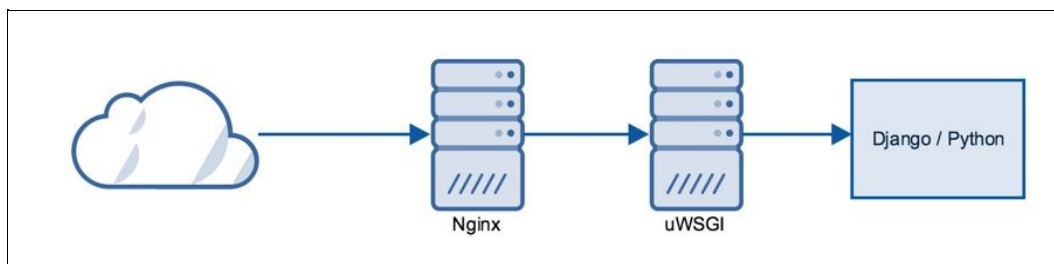


Getting ready

Django interfaces to NGINX through a **Web Server Gateway Interface (WSGI)**. For Django, one of the most commonly used WSGI interfaces is uWSGI.

If you haven't installed uWSGI yet, the best way is to install the latest version via `pip`: **`apt-get install python-pip python-dev`**
`pip install uwsgi`

We're going to simply use a base install of the latest version of Django, which at the time of writing this book was 1.10.5.



[uwsgi]

socket = 127.0.0.1:8000

uid=www-data

gid=www-data

chdir=/var/www/djangodemo module=djangodemo.wsgi master=True

pidfile=/tmp/uwsgi-djangodemo.pid vacuum=True

max-requests=5000

daemonize=/var/log/uwsgi/djangodemo.log

server {

listen 80; server_name djangodemo.nginxcookbook.com;

access_log /var/log/nginx/djangodemo-access.log combined;

location = /favicon.ico { access_log off; log_not_found off; }

location /static|/media {

root /var/www/djangodemo/; }

location / {

include uwsgi_params; uwsgi_pass 127.0.0.1:8000; }

}

How it works....

The first configuration item we tweak is the `favicon`:

```
| location = /favicon.ico { access_log off; log_not_found off; }
```

There's a very famous story of Instagram's first deployment (where they had 10,000 users in the first day) and the load that a missing `favicon` generated (since Django had to produce the 404 error) caused significant scaling issues.

Next, we serve any of the uploaded and static media directly via NGINX:

```
location /static|/media { root /var/www/djangodemo/; }
```

These directories are mapped via the `STATIC_ROOT` and `MEDIA_ROOT` configuration lines within the `settings.py` file for Django. NGINX is very efficient at serving static media, so serving it directly produces as little overhead as possible.

We then map app's all other URL calls via the `uwsgi` protocol: `location / { include uwsgi_params; uwsgi_pass 127.0.0.1:8000; }`

The uWSGI project has a native protocol (called `uwsgi` and in lower case), which is built into NGINX by default. It's a binary protocol designed to be highly efficient and scalable.

See also

For more information on the issues and best practices refer to the following link:

<http://uwsgi-docs.readthedocs.org/en/latest/ThingsToKnow.html>

Setting up NGINX with Express

Rather than being overly opinionated, Express is a very minimalistic and flexible framework suited for web and mobile application deployments based on Node.js. It's also one of the most popular bases for some of the more complex and feature-packed Node.js frameworks too, which is why it makes a great learning block for Node.js web deployments.



```
var express = require('express'); var app = express();
```

```
var expressWs = require('express-ws')(app);
```

```
app.get('/', function (req, res) {  
    res.send('Nginx demo!!!'); });
```

```
app.ws('/echo', function(ws, req) {  
    ws.on('message', function(msg) {  
        ws.send(msg); }); });
```

```
app.listen(3000);
```

```
server {  
  
    listen 80; server_name express.nginxcookbook.com;  
  
    access_log /var/log/nginx/express-access.log combined;  
  
    location / {  
  
        proxy_pass http://127.0.0.1:3000; proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade; proxy_set_header  
        Connection "upgrade"; }  
  
}
```

How it works....

There's only one main block directive to get our simple test working, with a few key areas.

```
| proxy_pass http://127.0.0.1:3000;
```

The proxy will forward the connection back to our Express-driven test application, which we configured to listen on port 3000 of the localhost.

```
| proxy_http_version 1.1;
```

By default, proxied connections back to the Node.js application will be HTTP/1.0 only. Setting this to HTTP/1.1 allows the use of keep-alive, which means the connection between NGINX and the application remains open rather than establishing a new connection every time. On a heavily loaded system, this is much more efficient.

```
| proxy_set_header Upgrade $http_upgrade;  
| proxy_set_header Connection "upgrade";
```

The WebSocket protocol uses the `Upgrade` and `Connection` headers as part of the handshake process. We need to set this at the NGINX level in order to allow the handshake process to work as expected. As it's compatible with HTTP/1.1, it won't interfere with the standard connections either.

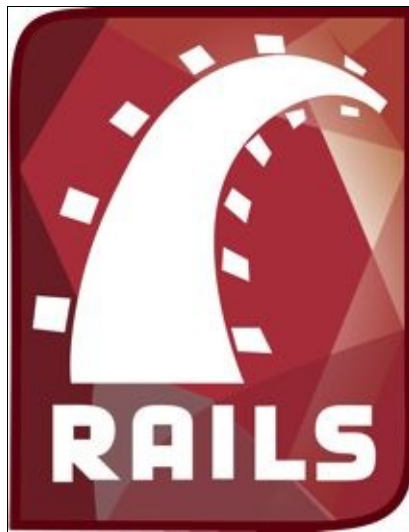
See also

For more information on WebSocket and the HTTP/1.1 Upgrade header refer to following links:

- <https://www.nginx.com/blog/websocket-nginx/>
- https://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header

Running Ruby on Rails with NGINX

Ruby on Rails is a full web application framework, based on the Ruby programming language. As with a more traditional framework such as Django, Rails is based around a standard **Model View Controller (MVC)** pattern. This makes it well suited to rapid development which is also highly performant and feature packed.



Getting ready

Although there are a number of different application servers for Rails, one of the more popular is Passenger. With the highest performance and great documentation, Passenger's popularity is well deserved. This recipe has been tested with Ruby 2.3, Rails 4.2, and Passenger 5.0.

We can install Passenger via `gem`:

```
| gem install passenger
```

To integrate with NGINX, this requires a special module to be enabled. We'll compile NGINX from scratch as per our installation in [Chapter 1, Let's Get Started](#) and compile Passenger support as a static module.



Make sure you have the mainline repository from nginx.org installed.

For a Debian- / Ubuntu-based system, you'll need to create the build environment:

```
| apt-get build-dep Nginx  
| mkdir ~/nginxbuild  
| cd ~/nginxbuild  
| apt-get source nginx
```

Then, in the NGINX source directory, edit the `debian/rules` file to add:

```
| --add-module=/var/lib/gems/2.3.0/gems/passenger-5.0.27/src/nginx_module \
```

You can confirm the `passenger` directory with the following command:

```
| passenger-config --nginx-addon-dir
```

To make the package name unique, you can edit the changelog and add additional lines to the changelog. Consider this example:

```
| nginx (1.9.14-1-passenger) wily; urgency=low  
|  
| * Added Passenger as a dynamic module
```

You can now compile the package with the following command:

```
| fakeroot debian/rules binary
```

This will take a few minutes to complete. Once we have the updated package, we can now install it:

```
| dpkg -i ../nginx_1.9.14-1-passenger_amd64.deb
```

passenger start --debug-nginx

```
include '/var/lib/gems/2.3.0/gems/passenger-5.0.27/resources/mime.types'; passenger_root '/var/lib/gems/2.3.0/gems/passenger-5.0.27'; passenger_abort_on_startup_error on; passenger_ctl cleanup_pidfiles L3RtcC9wYXNzZW5nZXItc3RhbmRhbg9uZS41aDBnZG0vdGVtcF passenger_ctl integration_mode standalone; passenger_ctl standalone_engine nginx; passenger_user_switching off;
```

```
passenger_ruby /usr/bin/ruby2.3;
```

```
passenger_user www-data;
```

```
passenger_default_user www-data;
```

```
passenger_analytics_log_user www-data; passenger_log_level 3;
```

```
server {
```

```
    server_name railsdemo.nginxcookbook.com; listen 80;
```

```
    access_log /var/log/nginx/rails-access.log combined; root /var/www/railsdemo/public; passenger_app_root /var/www/railsdemo; passenger_enabled on;
```

```
    location ~ ^/assets/ {
```

```
    }
```

```
}
```

```
passenger_pre_start http://0.0.0.0:3000/;
```

How it works...

There are a number of directives which aren't seen elsewhere because this recipe uses a specific Passenger module. In the core NGINX configuration file, we set the generic Passenger variables to use the correct Ruby version as well as the user which will run the Passenger server.

In our `server` block directive, we point Passenger to our demo Rails instance and enable Passenger for the server instance. There's also a basic *assets* location block directive, so that NGINX will process the static media natively.

Lastly, we call `passenger_pre_start` to automatically start the Passenger application server when NGINX starts. This saves you having to start it separately, as is the case for most other application servers.

See also

For more information on the configuration and optimization of Passenger server, refer to following link:

<https://www.phusionpassenger.com/library/config/nginx/reference/>

Easy Flask with NGINX

As one of the more popular Python frameworks, Flask is an extremely lightweight yet powerful system for web applications. Seen as more of a *micro* framework, a functional Flask application can be as simple as 20 lines of code.



```
from flask import Flask application = Flask(__name__)
```

```
@application.route("/") def hello():
```

```
    return "<h1>Demo via Nginx with uWSGI!</h1>"
```

```
if __name__ == "__main__": application.run(host='127.0.0.1',  
port=9001)
```

```
<strong>apt-get install python-pip python-dev<br/></strong>
```

```
<strong>pip install uwsgi</strong>
```

```
uwsgi --ini uwsgi.ini
```

2. We now need to configure the NGINX server block directive and again I've included this in a separate file (/etc/nginx/conf.d/flask.conf):

```
server {
    listen 80;
    server_name flaskdemo.nginxcookbook.com;
    access_log /var/log/nginx/flaskdemo-access.log com
    location = /favicon.ico { access_log off; log_not_
    location / {
        include uwsgi_params;
        uwsgi_pass 127.0.0.1:8000;
        uwsgi_param SCRIPT_NAME '/';
    }
}
```

How it works...

The Flask applications are so simple that they don't come with all of the boilerplate WSGI configurations such as Django and other frameworks. The good news is that the flexibility of uWSGI means it's very easy to configure.

In order for uWSGI to know what file to call, which we do via the `module` configuration entry, set it to `demoapp`. Those familiar with Python will know that this translates into loading `demoapp.py`.

Secondly, the `callable` configuration specifies the default WSGI callable name, which in our `demoapp.py` is set to `application`.

Lastly, we have a very simple NGINX `server` block directive. This simply pushes all calls to our Flask application using the uWSGI socket.

Laravel via NGINX

Inspired by the CodeIgniter framework, Laravel is a modern PHP framework based on the MVC pattern. Focusing on elegant code and modular packaging, Laravel is an increasingly popular choice for PHP developers who want to develop custom web applications.

In this recipe, we'll go through the code required to publish your Laravel application using NGINX.



Getting ready

This recipe assumes you already have a working Laravel installation as well as a working PHP-FPM installation. The following configuration has been tested with Laravel 5.2.

```
server {  
  
    listen 80;  
  
    server_name laravel.nginxcookbook.com; access_log  
/var/log/nginx/laravel.access.log combined; index index.php;  
  
    root /var/www/vhosts/laraveldemo/public;  
  
    location / {  
  
        try_files $uri $uri/ /index.php?$args; }  
  
    location ~ \.php$ {  
  
        fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index  
index.php; fastcgi_param SCRIPT_FILENAME <br/>  
$document_root$fastcgi_script_name; include fastcgi_params; }  
  
}
```

How it works...

Laravel stores the files accessible by PHP-FPM and NGINX in the public folder, not the root of the Laravel project. Make sure you point the root to the correct directory, otherwise you could expose your configuration files.

We set a `location` block (`location ~ \.php$`) so that only PHP files are compiled by our PHP-FPM configuration. This means that all other files (for example, CSS stylesheets) are served directly by NGINX for the greatest efficiency.

See also

Official Laravel site: <https://laravel.com/>

Meteor applications with NGINX

Meteor is a full-stack, JavaScript driven framework designed to rapidly build cross-platform applications. It is integrated with the in-house designed **Distributed Data Protocol (DDP)** and allows for seamless data synchronization to clients; it also includes an inbuilt MongoDB data store.



Getting ready

This recipe is based on Meteor 1.3 using a basic application with the Blaze templating engine.

meteor run

server {

listen 80; server_name meteorapp.nginxcookbook.com;

access_log /var/log/nginx/meteor-access.log combined;

location / {

proxy_pass http://127.0.0.1:3000; proxy_http_version 1.1;
proxy_set_header Upgrade \$http_upgrade; proxy_set_header
Connection "upgrade"; }

}

How it works...

Like the Express example, we use a proxy to the application and then set the headers (`Connection / upgrade`) to ensure WebSockets are passed through correctly.

High speed Beego with NGINX

Although Go is a relatively new programming language, its popularity has been rapidly increasing as the demand for higher performance and greater scalability increases. Go is derived from Google's desire to scale rapidly, yet safely, and already has a number of web frameworks available.

One of the most popular frameworks is Beego, an easy to use MVC-based web framework with integrated REST and code hot compile. By harnessing the performance of Go, Beego can outperform other frameworks (using a different programming language) by more than ten times in many instances.



Getting ready

In this recipe we're going to use the example `todo` code. This is available at <https://github.com/beego/samples/tree/master/todo>.

It's important to build this rather than just using `go run`, otherwise you'll have issues. Consider the following example:

```
cd $GOPATH/src/github.com/beego/samples/todo
go build
./main
```

```
server {  
  
    listen 80; server_name beegodemo.nginxcookbook.com; access_log  
/var/log/nginx/beegodemo-access.log combined; location =  
/favicon.ico { access_log off; log_not_found off; }  
  
    location / {  
  
        proxy_pass http://127.0.0.1:8080; proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade; proxy_set_header  
        Connection "upgrade"; }  
  
}
```

By default, a Beego application will listen on the localhost to port 8080. We simply proxy all connections though. Like the Express recipe, we also set the proxy headers to allow WebSockets to pass through as well.

All About SSLs

In this chapter, we will cover the following recipes:

- Basic SSL certificates
- Enabling HTTP/2 on NGINX
- Configuring HSTS in NGINX
- Easy SSL certificates with Let's Encrypt
- Making NGINX PCI-DSS compliant
- OCSP stapling with NGINX
- Achieving full A+ Qualys rating

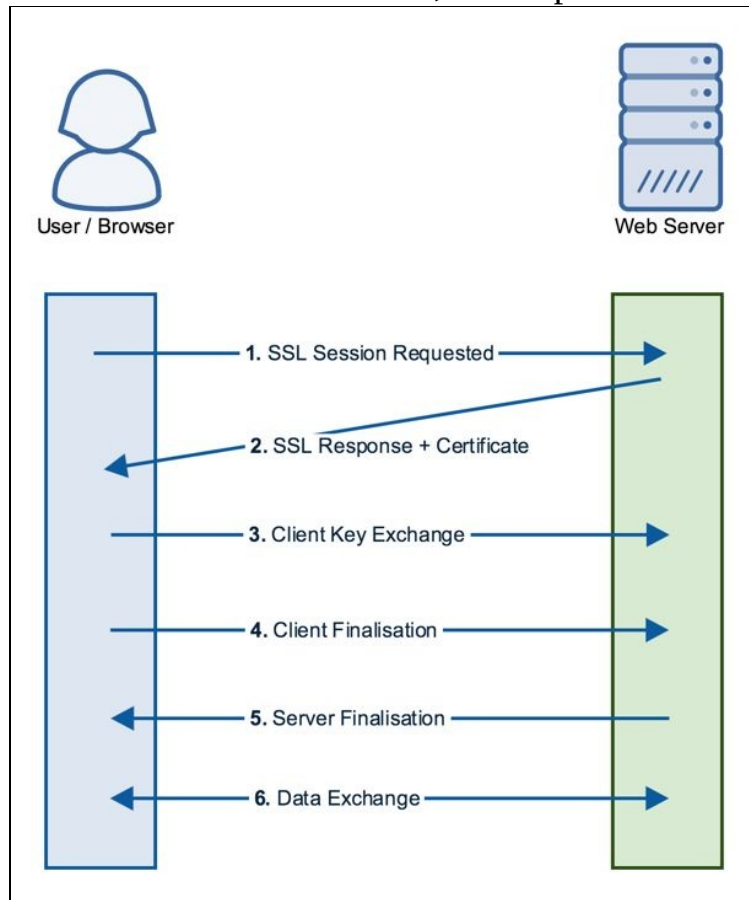
Introduction

The **Secure Sockets Layer (SSL)** standard has been traditionally used to encrypt web traffic that needs to be protected, such as financial transactions (for example, credit card numbers) and sensitive information. Recently, however, we have seen an ever-increasing trend of encrypting whole sites and all related services. The use of SSLs ensures that the whole transport of information is encrypted and therefore can't be intercepted, especially, now that Google has given a small **Search Engine Optimization (SEO)** ranking boost to sites that have an SSL enabled by default. While the boost is small, Google's focus on encouraging the safe transmission of data means that this will likely increase in the future.

Thankfully, NGINX's support for the latest SSL standards and the latest transport standards, such as HTTP/2 (covered in detail later), means that it's getting easier to efficiently deploy SSL-encrypted sites.

Basic SSL certificates

Before we get started, let's just have a refresher on how the browser-to-server encryption works and what we need to consider. This is a very brief overview specific to a basic web server scenario, so the process can vary for different



scenarios:

Following are the steps that happen in a web server scenario:

1. First, the browser communicates with the web server and requests the start of an SSL handshake. This is also where the browser can let the server know what cipher (encryption) algorithms it will allow.
2. Next, the server responds to the browser. At this stage, the server will confirm which cipher (based on the list provided by the browser) will be used. The server will also send a copy of the public certificate to the client. The browser will then communicate with the **Certificate Authority (CA)**

to authenticate the certificate.

3. Next, the key exchange is kicked off. A session key is established. This key is based on the public key on the client side and decoded by the private key on the server side.



It's important to note that the private key is never transmitted; it always remains on the server.

4. Once the session key is complete, the client will send a final confirmation to complete the handshake and await a reciprocal finalization from the server side.
5. Finally, we have a secure tunnel in which encrypted data can now be transmitted. This is where the actual web content can now be sent.

Getting ready

To install an SSL, there are three components we need to start with. The first is **Certificate Signing Request (CSR)**. This defines the information which will be contained within the certificate and includes things such as the organization name and domain name. The CSR is then sent to a CA or used to generate a self-signed certificate.

To make it easy for this recipe, we'll use a self-signed certificate. We can easily generate the CSR and then the private key and public certificate with one command. For example, here's how to generate a CSR with a 2048 bit key and 600 day expiry: **openssl req -x509 -new -newkey rsa:2048 -nodes -keyout private.key -out public.pem -days 600**

This example will ask a series of questions for the CSR and then automatically generate the private key (`private.key`) and the public certificate (`public.pem`). Consider the following example:

```
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'private.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:AU
State or Province Name (full name) []:QLD
Locality Name (eg, city) [Default City]:Brisbane
Organization Name (eg, company) [Default Company Ltd]:Nginx Cookbook
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:ssl.nginxcookbook.com
```



Self-signed certificates aren't validated by browsers and are not intended for production. They should be used for internal and testing purposes only.

How to do it...

Now that we have a certificate and private key, we can update our NGINX configuration to serve SSL-based sites. Here's our NGINX `server` directive block:

```
server {  
    listen          443 ssl;  
    server_name     ssl.nginxcookbook.com;  
    ssl_certificate  /etc/ssl/public.pem;  
    ssl_certificate_key /etc/ssl/private.key;  
    ssl_protocols   TLSv1 TLSv1.1 TLSv1.2;  
    ssl_ciphers     HIGH:!aNULL:!MD5;  
  
    access_log /var/log/nginx/ssl-access.log combined;  
  
    location /favicon.ico { access_log off; log_not_found off; }  
    root /var/www;  
}
```

If you have a basic `index.html` or similar in `/var/www`, you should see something like the following:



The error message will vary between browsers, but they're all simply letting you know that the certificate presented couldn't be validated and therefore can't be intrinsically trusted. For testing, add an exception here; you should see the SSL site served by NGINX:



How it works...

Here's what the directives do:

- `listen 443 ssl`: Firstly, we tell NGINX to listen on port 443 (the HTTPS standard) using the SSL protocol. Previously, we'd simply told NGINX to listen on port 80, and it used HTTP by default.
- `ssl_certificate`: This is the location of the public key, which needs to be in the PEM format. If your CA also provided intermediate certificates, then they also need to be in this file.
- `ssl_certificate_key`: This is the location of the private key, and it also needs to be in PEM format. This key needs to be kept safe to ensure the integrity of your certificate—it should only reside on the server.
- `ssl_protocols`: Here, we specify what variants of the SSL protocols we want to make available. The easiest default is to support **Transport Layer Security (TLS)**, which is the successor to the older SSL protocol. As both SSLv2 and SSLv3 have had significant flaws exposed in recent years, they should only be enabled as a last resort.
- `ssl_ciphers`: The ciphers dictate the type of encryption used and at what level. The default of `HIGH:!aNULL:!MD5` means that we use only high grade (128-bit and higher), authenticated (the exclamation means NOT) encryption and not MD5 hashing. The defaults are secure and shouldn't be changed without a good reason.

Enabling HTTP/2 on NGINX

The now ratified HTTP/2 standard is based on SPDY, an experimental protocol that Google developed internally. As shown in the diagram in the previous recipe, establishing an HTTPS connection can be quite time consuming. With HTTP/1.1, each connection to the web server must follow this process and wait for the handshake to complete.

In HTTP/2, this handshake time is reduced, but more importantly the requests are multiplexed over a single TCP connection. This means that the handshake only has to occur once, significantly reducing the latency of a site for the end user. In fact, it means that an HTTP/2-based site can actually be quicker than a standard HTTP-based one.

There are a number of other benefits that HTTP/2 also provides, such as header compression, a new binary protocol, and a server-based push. All of these further increase the efficiency of HTTP/2, yet it also remains backwards compatible with HTTP/1.1:

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android *	Chrome for Android
			² 49						
			^{2.4} 55			² 9.3		4.4	
	² 14	² 51	^{2.4} 56	^{2.3} 10	^{2.4} 43	² 10.2		4.4.4	
^{1.2} 11	² 15	² 52	^{2.4} 57	^{2.3} 10.1	^{2.4} 44	² 10.3	all	² 56	^{2.4} 57
		² 53	^{2.4} 58	^{2.3} TP	^{2.4} 45				
		² 54	^{2.4} 59		^{2.4} 46				
		² 55	^{2.4} 60						

HTTP/2 support

Source: caniuse.com (April 2017)

All modern browsers (as shown in the preceding figure) support HTTP/2 natively, so it's ready to deploy in production. NGINX officially supported HTTP/2 starting with version 1.9.5.

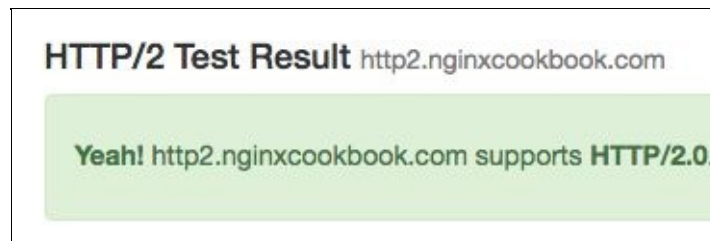
How to do it...

Based on our previous example, we only need to make one key change. Instead of just specifying `ssl` as the listening protocol, we add `http2`:

```
server {  
    listen          443 ssl http2;  
    server_name     http2.nginxcookbook.com;  
    ssl_certificate  /etc/ssl/public.pem;  
    ssl_certificate_key /etc/ssl/private.key;  
    ssl_protocols   TLSv1 TLSv1.1 TLSv1.2;  
    ssl_ciphers     HIGH:!aNULL:!MD5;  
  
    access_log /var/log/nginx/ssl-access.log combined;  
  
    location /favicon.ico { access_log off; log_not_found off; }  
    root /var/www;  
}
```



*No changes to protocols or ciphers are necessary. To verify that HTTP/2 is working, you can use Chrome **Developer Tools** (**DevTools**) to show the protocol, or an external tool, such as the one KeyCDN provides:*



Source: <https://tools.keycdn.com/http2-test>

While a simple page won't show the performance difference, with the average page demand of around 50–70 requests we should see about 20 percent improvement. For more complex sites (for example, e-commerce sites) where there are more than 200 requests, the difference is even more dramatic.

See also

NGINX HTTP/2 white paper: https://assets.wp.nginx.com/wp-content/uploads/2015/09/NGINX_HTTP2_White_Paper_v4.pdf

Configuring HSTS in NGINX

HTTP Strict Transport Security (HSTS) is an enhancement to the HTTP protocol that is designed to enforce strict security standards to help protect your site and users. HSTS does a number of things. Firstly, it ensures that all requests must be made via HTTPS. This ensures that data isn't accidentally sent via HTTP and, therefore, left unencrypted.

Secondly, it ensures that only a valid certificate can be accepted. In our previous examples, we used a self-signed certificate and the browser allowed us to bypass the security check. With HSTS enabled, this is no longer possible. This means that attempts to emulate your site or man-in-the-middle attacks where a different certificate is used are now no longer possible.

```
server {  
  
    listen 443 ssl http2; server_name http2.nginxcookbook.com;  
    ssl_certificate /etc/ssl/public.pem; ssl_certificate_key  
    /etc/ssl/private.key; ssl_protocols TLSv1 TLSv1.1 TLSv1.2;  
    ssl_ciphers HIGH:!aNULL:!MD5;  
  
    add_header Strict-Transport-Security "max-age=31536000;  
  
    access_log /var/log/nginx/ssl-access.log combined;  
  
    location /favicon.ico { access_log off; log_not_found off; }  
  
    root /var/www;  
  
}
```

This header specifies the max-age, which in our case we have set to 31536000 seconds (which is 365 days). This means that the browser will cache the HSTS settings for an entire year and ensure that all requests for the next 365 days will be HTTPS only for your domain. It will also only accept the official certificate.

```
add_header Strict-Transport-Security "max-age=31536000;"
```

```
location /api/ {
```

```
    add_header 'Access-Control-Allow-Origin' '*'; add_header 'Access-  
Control-Allow-Methods' 'GET, POST, <br/> OPTIONS'; add_header  
    Strict-Transport-Security "max-age=31536000;" }
```

As we added access control headers, we needed to redeclare the HSTS header.

Easy SSL certificates with Let's Encrypt

Let's Encrypt is a free, automated CA, which is made possible due to the sponsorship from companies such as the **Electronic Frontier Foundation (EFF)**, Mozilla, Cisco, Akamai, University of Michigan, and over a dozen others. Organized by the **Internet Security Research Group (ISRG)**, Let's Encrypt has already issued over 4 million certificates (as of May 2016), and the rate is growing exponentially.

While the free component may see the biggest drawback, it's the automation that is the key point. For those who have previously gone through SSL generation through a traditional CA, it needs to have either file-based or email-based validation of the domain. If you have multiple domains, this process becomes quite time consuming to do over and over.

Thankfully, the Let's Encrypt CA is fully API-driven, and they even include a client to automate everything from the validation to the update of the NGINX configuration. There are also extensions for popular web control panels to automate the process as well, such as Plesk and CPanel.

How to do it...

The official Let's Encrypt client, recently renamed Certbot, is easy to use and install. The installation is the same for most distributions, and has been greatly simplified compared to the Let's Encrypt beta phase. Here's how to install it:

```
wget https://dl.eff.org/certbot-auto  
chmod a+x certbot-auto  
./certbot-auto
```

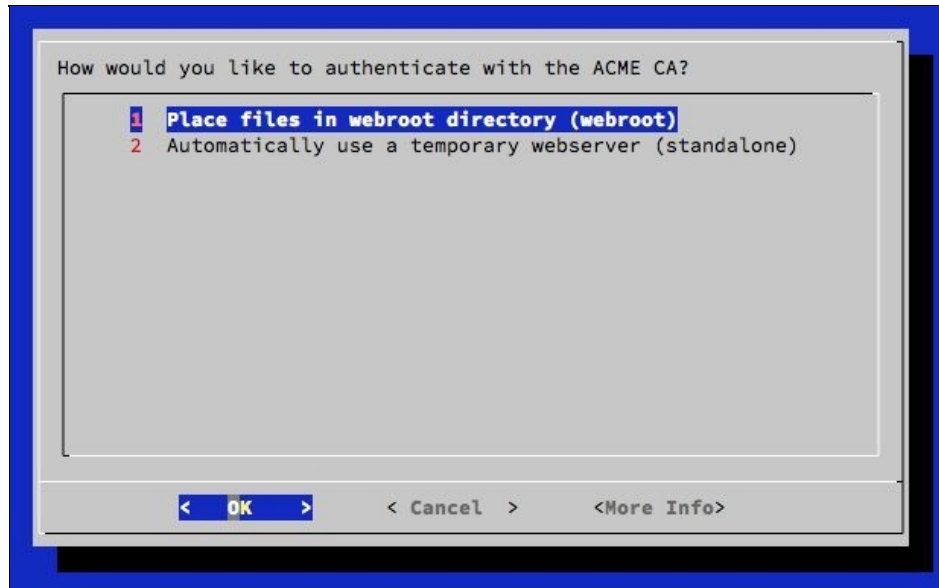


This will download and install all of the required Python packages for your system, then create the Certbot-specific **Virtual Environment (Virtualenv)**. Depending on the speed of your system, this may take up to five minutes to complete.

Unfortunately, the NGINX plugin to automatically create the configuration isn't complete yet, so we need to manually edit our configuration. First, let's generate the certificate:

```
| ./certbot-auto certonly
```

This will present us with two options for domain validation: It can be achieved either by placing files within your existing `webroot` directory or by running a temporary web server:



I find that the standalone web server is the most flexible option, especially if your NGINX configuration doesn't serve static files. However, it does require you to stop any services using port 80 first.

Next, the wizard will ask you for an email account. This is used to notify you when your certificate is about to expire or to notify you if it has been revoked. You'll then need to enter the domain name to generate the SSL certificate.



This domain has to be valid and the DNS pointed at the server for it to validate correctly.

If Certbot was able to validate, you should see a notice letting you know your certificate is ready:

```
IMPORTANT NOTES:
- Congratulations! Your certificate and chain have been saved at
  /etc/letsencrypt/live/letsencrypt.nginxcookbook.com/fullchain.pem.
  Your cert will expire on 2016-09-03. To obtain a new or tweaked
  version of this certificate in the future, simply run certbot-auto
  again. To non-interactively renew *all* of your certificates, run
  "certbot-auto renew"
- If you like Certbot, please consider supporting our work by:

  Donating to ISRG / Let's Encrypt:  https://letsencrypt.org/donate
  Donating to EFF:                   https://eff.org/donate-le
```

It's worth noting that while the expiry date looks short (90 days), it's designed to be automated for renewal. We can run the renewal process with the following command:

```
| ./certbot-auto renew
```

As this can be run noninteractively, you can set it to autorenew via a Cron call or similar.

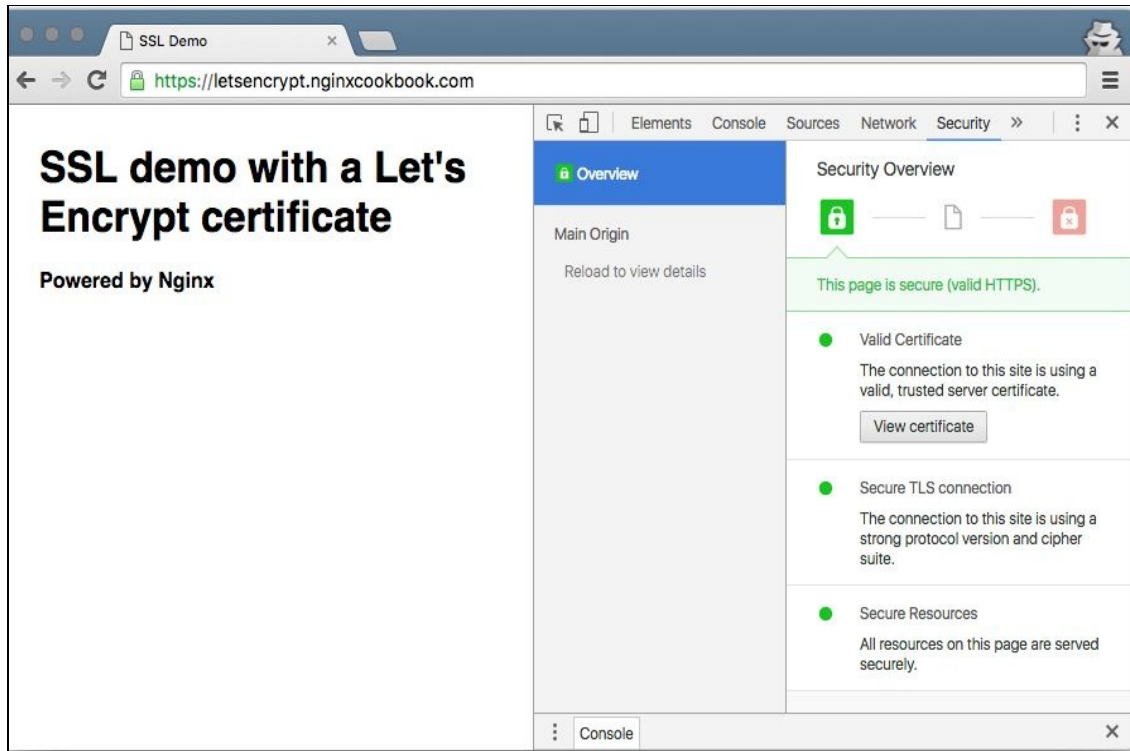
Now, we have a certificate pair (located in `/etc/letsencrypt/live/`), we can now create an NGINX configuration to use them:

```
server {
    listen            443 http2;
    server_name       letsencrypt.nginxcookbook.com;
    ssl_certificate    /etc/letsencrypt/live/letsencrypt.nginxcookbook.com
                      /fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/letsencrypt.nginxcookbook.com
                      /privkey.pem;
    ssl_protocols     TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers        HIGH:!aNULL:!MD5;

    access_log /var/log/nginx/letsencrypt-access.log combined;

    location /favicon.ico { access_log off; log_not_found off; }
    root /var/www;
}
```

With our new configuration loaded, we can verify that the certificate is valid:



As the Chrome DevTools show, the certificate used is validated by the browser and gives the green lock for confirmation. There's no difference in this certificate compared to other CAs, except you can automate the generation and renewal!

See also

The official Certbot site: <https://certbot.eff.org/>

Making NGINX PCI DSS compliant

The **Payment Card Industry Data Security Standard (PCI DSS)** is a set of 12 security standards designed to ensure the secure transmission and storage of payment-related information. These standards set out a stringent set of rules covering everything from server security to policy and business standards.

We'll focus only on one part of Requirement 4, which is entitled *Encrypt transmission of cardholder data across open, public networks*.

```
server {  
  
    listen 443 http2 default_server; server_name  
    pcidss.nginxcookbook.com; ssl_certificate <br/>  
    /etc/letsencrypt/live/pcidss.nginxcookbook.com/<br/> fullchain.pem;  
    ssl_certificate_key <br/>  
    /etc/letsencrypt/live/pcidss.nginxcookbook.com/<br/> privkey.pem;  
  
    ssl_protocols TLSv1.1 TLSv1.2; ssl_ciphers  
    HIGH:!aNULL:!MD5:!kEDH; ssl_prefer_server_ciphers on;  
    ssl_session_cache shared:SSL:10m;  
  
    add_header Strict-Transport-Security "max-age 31536000";  
  
    access_log /var/log/nginx/pcidss-access.log combined;  
  
    location /favicon.ico { access_log off; log_not_found off; }  
  
    root /var/www;  
  
}
```

How it works...

Let's look at some of the differences:

- `listen 443 http2 default_server;` We add the `default_server` so that NGINX has a default configuration it will use during the negotiation phase. This is because the **Server Name Indication (SNI)** only occurs after the connection has been negotiated. Without specifying the `default_server` directive, the initial handshake would revert to TLS 1.0 (the NGINX default).
- `ssl_protocols TLSv1.1 TLSv1.2;` We specify only the TLS 1.1 and 1.2 protocols, which disables the older 1.0 (vulnerable to POODLE attacks). This also ensures that the older SSLv2 and SSLv3 remain disabled as well.
- `ssl_ciphers HIGH:!aNULL:!MD5:!kEDH;` While this doesn't look much different to our previous examples, we have added the `!kEDH` parameter to the end. This disables the basic Diffie Hellman key exchange (vulnerable to LOGJAM) while still allowing the more efficient, **Elliptic curve Diffie-Hellman (ECDH)**.

With this combination, we're able to achieve an A+ rating with the High-Tech Bridge SSL Server Test, along with a confirmation that the settings are PCI DSS

compliant:



See also

- The official PCI-DSS site: <https://www.pcisecuritystandards.org>
- High-Tech Bridge SSL Server Test: <https://www.htbridge.com/ssl/>

OCSP stapling with NGINX

Online Certificate Status Protocol (OCSP) is one of the main protocols used for checking for revoked certificates. This is important in order to ensure that if a server or certificate was compromised, the certificates can be replaced and the old ones revoked to prevent fraudulent use.

These checks can be time consuming, as the browser has to validate the certificate the first time it's used. OCSP stapling is an alternative to OCSP, and alleviates some of the latency issues associated with OCSP. It does this by stapling a cached result directly to the main request. As this result is still signed by the CA, it means that the results are just as secure, yet with no additional latency.

How to do it...

In order to use OCSP stapling, we need to add two additional lines to our `server` directive: `ssl_stapling on; ssl_stapling_verify on;`

This means that the server is now responsible for the initial OCSP lookup and will then send every subsequent request with the cached result.

We can verify that this is working with the Qualys SSL Server Test, and we should be looking for the following:

OCSP stapling

Yes

CloudFlare (a content distribution and website performance platform) claims that OCSP stapling saves up to 30 percent of the SSL negotiation time, so enabling this is key for maximizing website performance.

See also

- The OCSP stapling Wikipedia page: https://en.wikipedia.org/wiki/OCSP_stapling
- Qualys SSL Server Test: <https://www.ssllabs.com/ssltest>

Achieving full A+ Qualys rating

One of the benchmarks for configuring an SSL-based site is to achieve an A+ rating using the Qualys Server SSL Test. This is because Qualys has set a stringent set of expected results and minimum standards, designed to ensure that your website is as secure as possible.

Achieving this requires you to disable old protocols and ciphers, much in the way we do for PCI-DSS configurations. In fact, the basic PCI-DSS configuration we tested earlier already achieves an A+ rating. We're going to take it and go a bit further to give the ultimate SSL configuration for NGINX.



Some of these changes can cause backwards compatibility issues with older devices and browsers. Ensure you test it against your intended target audience thoroughly before using it in production.

Before we start, here's what a basic configuration (for example, our Let's Encrypt recipe) achieves:



A B score isn't bad, and the main area it's downgraded for is the weak Diffie-Hellman key exchange. We'll ensure this isn't an issue in this recipe.

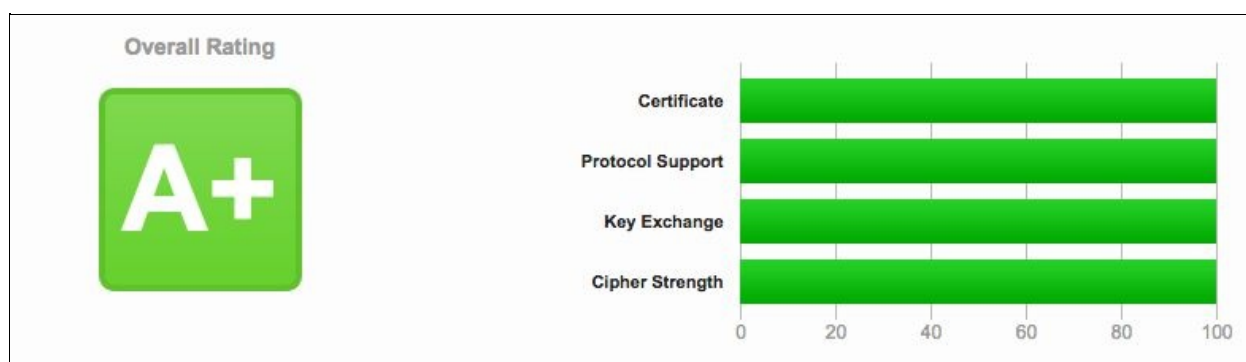
How to do it...

The first thing we need to do is regenerate our certificate to ensure that it's 4096-bit for the RSA key. Most CAs issue a 2048-bit certificate, which will only give us 90 percent for the Key Exchange evaluation. If you're using Let's Encrypt, you'll need to generate the certificate with the `--rsa-key-size 4096` parameter.

Here's our `server` directive, which has been based on the PCI-DSS configuration and tweaked further:

```
server { listen 443 http2 default_server; server_name
ultimate.nginxcookbook.com; ssl_certificate
/etc/letsencrypt/live/ultimate.nginxcookbook.com
/fullchain.pem; ssl_certificate_key
/etc/letsencrypt/live/ultimate.nginxcookbook.com
/privkey.pem; ssl_protocols TLSv1.2; ssl_ciphers
EECDH+CHACHA20:EECDH+CHACHA20-
draft:EECDH+AES128:RSA+AES128:EECDH+AES256:
RSA+AES256:EECDH+3DES:RSA+3DES:!MD5; ssl_prefer_server_ciphers
on; ssl_session_cache shared:SSL:10m; ssl_stapling on; ssl_stapling_verify on;
ssl_dhparam /etc/ssl/dhparam.pem; add_header Strict-Transport-Security "max-
age=31536000"; access_log /var/log/nginx/ultimate-access.log combined;
location /favicon.ico { access_log off; log_not_found off; } root /var/www; }
```

With this set, we achieve A+, as shown in the following image:



Using the High-Tech Bridge SSL Server Test we also achieve full NIST-

recommendation compliance as well:



To restate the note at the start, the more stringent the security rules, the greater the chance of issues with older systems and browsers.

How it works...

Firstly, we only allow TLS 1.2. Qualys to require this for the 100 percent protocol grading.

Next, we set a very limited number of ciphers, all of which are 256 bit or higher. We've also set it to use ECDH only, to enforce forward secrecy. This is combined with the 384-bit curve (`secp384r1`), which is the grade that the NSA mandate for top-secret graded documents. This is roughly the equivalent of a 7680-bit RSA key, so don't be fooled by the lower bit count.

See also

- Qualys SSL Server Test: <https://www.ssllabs.com/ssltest>
- Qualys Server Rating Guide: <https://github.com/ssllabs/research/wiki/SSL-Server-Rating-Guide>

Logging

In this chapter, we will cover the following recipes:

- Logging to syslog
- Customizing web access logs
- Virtual host log format
- Application focused logging
- Logging TLS mode and cipher information
- Logging POST data
- Conditional logging
- Using the Elastic Stack

Introduction

Akin to metrics, logging is key to monitoring how your server is performing and debugging errors. Thankfully, NGINX has an extensive logging module built-in to cover nearly all usage scenarios. In this chapter, we'll go through some of the various ways you can change the standard logging formats, as well as how to process them in more efficient manners.

Logging to syslog

If you already have your centralized server logs or your logs are analyzed by a standard syslog system, you can also redirect your NGINX logs to do the same. This is useful when using external platforms such as Loggly and Papertrail, which integrate via syslog.

```
server {
```

```
    listen 80; server_name syslog.nginxcookbook.com; access_log  
    syslog:server=unix:/dev/log; error_log syslog:server=unix:/dev/log;  
    location /favicon.ico { access_log off; log_not_found off; }
```

```
    root /var/www; }
```

```
<strong>tail -n 1 /var/log/syslog</strong>
```

```
<strong>Jun 22 23:40:53 nginx-220-ubuntu nginx-220-ubuntu nginx:  
<br/>106.71.217.155 - - [22/Jun/2016:23:40:53 +1000] "GET /  
HTTP/1.1" 200 <br/>192 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS  
X 10_11_4) <br/>AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/51.0.2704.103 Safari/537.36"</strong>
```

Remote syslog

While we can generate a bit more flexibility using a local syslog daemon, generally, the reason we want to offload the logging is to either decrease the load on our production systems or to use external tools for better log analysis.

The simplest way to do this is to send the logs to a syslog daemon or processing system that doesn't exist on the same server. This can also be used to aggregate logging from multiple servers, which facilitates monitoring and reporting in a central location.

To send the logs to a remote server, firstly, we need to enable the syslog server to listen on a network port. This is disabled on most `rsyslog`-based systems, so that it prevents accidental security issues. To enable, we simply uncomment the following in the `rsyslog.conf` (generally `/etc/rsyslog.conf`) file:

```
module(load="imudp") input(type="imudp" port="514")
```

After the syslog daemon is restarted (`systemctl restart rsyslog`), it will now be listening on port 514 for UDP log packets.



By default, syslog data is decrypted. We highly recommend that you ensure that the logs are sent via a VPN tunnel or similar encrypted means.

On the NGINX server, we now need to update the `server` block for the access and debug logs:

```
| access_log syslog:server=202.74.71.220,tag=nginx,severity=info combined;  
| error_log syslog:server=202.71.71.220,tag=nginx,severity=error debug;
```

After reloading NGINX to apply the new rules, we can now verify that the logs are hitting the syslog on the remote server:

```
| Aug 09 23:39:02 nginx-220-ubuntu nginx: 106.71.219.248 - - [09/Aug/2016:23:39:02 +1000]  
| Aug 09 23:39:06 nginx-220-ubuntu nginx: 106.71.219.248 - - [09/Aug/2016:23:39:06 +1000]
```

On a Debian- / Ubuntu-based system, this will be `/var/log/syslog`. If you run a RedHat- / CentOS-based system, this will be logged to `/var/log/messages`.

See also

NGINX reference: <https://nginx.org/en/docs/syslog.html>

Customizing web access logs

There are a number of ways you can customize the log files, including the format, the information provided, and where to save them to. This customization can be especially handy if you're using NGINX as a proxy in front of application servers, where traditional web style logs may not be as useful.

How to do it...

There are a few different ways we can customize the access logs in order to get more relevant information or reduce the amount of logging where it's not required. The standard configuration of `combined` is as follows: `log_format combined '$remote_addr - $remote_user [$time_local] ' '"$request" $status $body_bytes_sent ' '"$http_referer" "$http_user_agent"'`;

This format is already predefined in NGINX out of the box, which is compatible with the Apache combined format. What this means is that, by default, combined formatted logs will be compatible with most log parsers and, will be therefore, able to directly interpret the log data.

While having more data can be quite helpful (as we'll show later), be careful about deviating from a standard format. If you want to use a third-party system to parse the logs, you'll also need to update or customize the parser as well.

See also

- The NGINX log module documentation: http://nginx.org/en/docs/http/nginx_http_log_module.html
- The NGINX core variables: http://nginx.org/en/docs/http/nginx_http_core_module.html#variables

Virtual host log format

If you're running a virtual host style environment (with multiple `server` blocks) with NGINX, there's one small tweak you can make to enhance the logs. By default, the host (defined as `$host`) isn't logged when using the default combined format. Having this field in the logs means that the log can be parsed externally without the need for additional information.

How to do it...

To start with, we need to define our new log format:

```
log_format vhostlogs '$host $remote_addr - $remote_user '
                    '[$time_local] "$request" $status '
                    '$body_bytes_sent "$http_referer" '
                    '"$http_user_agent"';
```



The `log_format` directive needs to be outside the `server` block.

To use our new log file, we update the access log format in the `server` block with the new format. Here's an example with our MediaWiki recipe:

```
| access_log /var/log/nginx/mediawiki-vhost-access.log vhostlogs;
```

If you had a wildcard setup for your subdomains, this would mean that you'd also have the correct hostname in the logs as well. Without this, the logs wouldn't be able to differentiate between any of the subdomains for statistical analysis and comparison.

With the updated format, here's an example from our logs:

```
| mediawiki.nginxcookbook.com 106.71.219.248 - - [06/Aug/2016:21:34:42 +1000] "GET /wiki/
```

We can now see the hostname clearly in the logs.

Application focused logging

In this recipe, we're going to customize the logs to give us a bit more information when it comes to applications and metrics. Additional information, such as the response time, can be immensely useful for measuring the responsiveness of your application. While it can generally be generated within your application stack, it can also induce some overhead or give incomplete results.

How to do it...

To start with, we need to define our new log format:

```
log_format applogs '$remote_addr $remote_user $time_iso8601'
                    '"$request" $status $body_bytes_sent '
                    '$request_time $upstream_response_time';
```



The `log_format` directive needs to be outside the `server` block.

Here, we've deviated from the combined format to change the time to the ISO 8601 format (which will look something like `2016-07-16T21:48:36+00:00`), removed the HTTP referrer and user agent, but added the request processing time (`$request_time`) to get a better idea on how much time it's taking our application to generate responses.

With our new log format defined, we can now use this for our application logs:

```
| access_log /var/log/nginx/meteor-access.log applogs;
```

We're using the default Meteor application which we had set up in [Chapter 3, Common Frameworks](#) and as we want applications to be very responsive, `$request_time` will provide instant feedback as to what calls aren't quick enough. Here's an example output from the logs:

```
| 106.70.67.24 - 2016-08-07T20:47:07+10:00"GET / HTTP/1.1" 200 2393 0.005
| 106.70.67.24 - 2016-08-07T20:47:07+10:00"GET /sockjs/070/tsjnlv82/websocket HTTP/1.1" 1
```

As the request times are measured in milliseconds, we can see that our base call took 0.005 ms to complete and the WebSocket connection took 14 ms to complete. This additional information can now be easily searched for and, with additional log parsers (like the Elastic Stack detailed later in this chapter), we can set further searches and alerts.

Logging TLS mode and cipher information

With the advent of HTTP/2 and the ever changing cryptography best practices, small compatibility issues can arise, which are very difficult to resolve. Browsers also change what they accept on a constant basis. To ensure, we know exactly what ciphers have been used with what protocol, we can add this additional information to our log files.

```
log_format ssl_logs '$remote_addr - $remote_user [$time_local] '
```

```
    '$request' $status $body_bytes_sent '
```

```
    '$http_referer' '$http_user_agent'
```

```
    '[$ssl_protocol|$ssl_cipher]';
```

```
106.70.67.24 - - [07/Aug/2016:22:39:20 +1000] "GET / HTTP/2.0"  
304 118 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5)  
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2490.71  
Safari/537.36"[TLSv1.2|ECDHE-RSA-AES128-GCM-SHA256]
```

```
106.70.67.24 - - [07/Aug/2016:22:39:20 +1000] "GET / HTTP/2.0"  
304 118 "-" "Mozilla/5.0 (Linux; Android 5.1.1; Nexus 6  
Build/LYZ28E) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/44.0.2403.20 Mobile Safari/537.36"[TLSv1.2|ECDHE-RSA-  
AES128-GCM-SHA256]
```

```
106.70.67.24 - - [07/Aug/2016:22:39:29 +1000] "GET / HTTP/2.0"  
304 118 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5)  
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103  
Safari/537.36"[TLSv1.2|ECDHE-RSA-AES128-GCM-SHA256]
```

```
106.70.67.24 - - [07/Aug/2016:22:42:31 +1000] "GET / HTTP/1.1"  
200 215 "-" "curl/7.40.0"[TLSv1.2|ECDHE-RSA-AES128-GCM-  
SHA256]
```

In the log details, we can now see the requests from Chrome, Firefox, Safari, and even cURL are using TLSv1.2 and ECDHE-RSA-AES128-GCM-SHA256. If we saw abnormal behavior with a different browser or protocol, this would greatly assist in the diagnosis of the problem.

Logging POST data

When we have form submissions, errors in this become difficult to replay and debug if we don't know the value of the data. By default, NGINX doesn't log POST data, as it can be very bulky. However, there are certain situations where getting access to this data is vital to debugging issues.

How to do it...

In order to log POST data, we need to define a custom log format:

```
| log_format post_logs '[$time_local] "$request" $status '
|                        '$body_bytes_sent "$http_referer" '
|                        '"$http_user_agent" [$request_body]';
```

By logging `$request_body`, we'll be able to see the contents of a POST submission.

To enable our POST log format, all we need to do is specify the log format for the access logs:

```
| access_log /var/log/nginx/postdata-access.log post_logs;
```

If you just want to verify that it's working as expected, we can use cURL to send post data. I've used Express, as set up in [Chapter 3, Common Frameworks](#), to accept the POST data, so we can make a call as follows:

```
| curl http://express.nginxcookbook.com/ -d 'name=johnsmith,phone=123123,email=john@smith
```

This posts the variables name, phone, and email through to the root (/) URL. If we look at `postdata-access.log`, we can now see the following:

```
| [09/Aug/2016:22:45:35 +1000] "POST / HTTP/1.1" 200 18 "-" "curl/7.43.0" [name=johnsmith,
```

If you still had the standard log format enabled, it would simply display a dash (-) for the request, making further debugging impossible if it wasn't captured by the application.

Here's the corresponding log entry in a standard combined format:

```
| 106.70.67.24 - - [09/Aug/2016:22:45:35 +1000] "POST / HTTP/1.1" 200 18 "-" "curl/7.43.0
```

Of course, by logging all of the POST data, you'll use considerably more disk space and server overhead. We highly recommend that this is only enabled for development environments.

Conditional logging

There are some instances where we may want to log different data, based on some sort of conditional argument. For instance, if you have a beta copy of your application and are interested in gathering more information for it, we can log this data just for the beta URL. Rather than clogging up our log files with the additional information when it's not required, we can simply trigger it when required.

How to do it...

To enable, we map the (\$args) URI arguments to a \$debuglogs variable, based on if a condition is set. Here's the code:

```
map $args $debuglogs {  
    default    0;  
    debug      1;  
}
```

Like the custom log formats, this needs to be placed outside of the `server` directives. Then, within the `server` directive, we can make use of the variable and create additional logging:

```
access_log /var/log/nginx/djangodemo-debug-access.log applogs if=$debuglogs;  
access_log /var/log/nginx/djangodemo-access.log combined;
```

If we call a normal URL, it will simply log the access to the standard `djangodemo-access.log` in the combined format. If we call a URL with the `debug` argument set (for example: `http://djangodemo.nginxcookbook.com/?debug`), we now get the details logged in both the standard logs as well as our additional `djangodemo-debug-access.log` using the `applogs` format we defined in the previous recipe.

Using the Elastic Stack



Manually digging through log files to gain insights or to detect anomalies can be very slow and time-consuming. To solve this issue, we're going to run through a quick example using the Elastic Stack (previously referred to as the **Elasticsearch Logstash Kibana (ELK)** stack. Elasticsearch is a high-speed search engine which offers real-time indexing and searching. Data is stored as schema-less JSON documents and has an easy-to-use API for access.

To complement this, there's also Logstash and Kibana. Logstash is a tool that allows for the collection of logs, parses the data on the fly, and then pushes it to a storage backend such as Elasticsearch. The original creator of Logstash (Jordan Sissel) wrote it to parse web server log files such as those produced by NGINX, so it's well suited to the task.

Kibana is then the final part of the puzzle. Once the data has been collected, parsed, and stored, we now need to be able to read it and view it. Kibana allows you to easily visualize the data by allowing you to easily structure the filters and queries and then display the information in easy-to-read graphs and tables. As a graphical representation, it is far quicker and easier to interpret and see variances in data which may be meaningful.

What's more, the whole ELK stack is open source and therefore freely available to use and extend. In fact, the stack itself has expanded with tools such as Beats to provide lightweight data shipping to be processed by Logstash. Once you start using it, it's hard to revert to flat files!

How to do it...

To run the Elastic Stack, you can either install it on your own systems or use a cloud-based instance. For this recipe, we're going to install it on a separate server.

Elasticsearch

To install Elastic, simply download the appropriate `.deb` or `.rpm` package for your particular instance. For the example, I'll use Ubuntu 16.04. First, we'll need to install Java version 8 (required for Elastic 5.x):

```
| apt install openjdk-8-jre
```

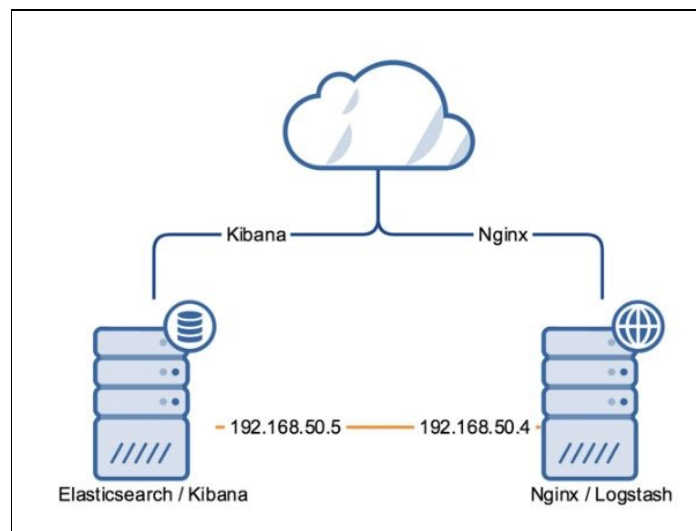
Next, we'll download the latest Elastic package and install it:

```
| wget https://download.elastic.co/elasticsearch/release/org/elasticsearch/distribution/d
| dpkg -i elasticsearch-5.0.0-alpha4.deb
```



The example is also using the 5.x release, which was in alpha during writing. Make sure you select the latest stable release for production use.

While Elasticsearch can become quite a complex system as you start to use it heavily, the good news is that the default configuration doesn't require any changes to get started. However, we highly recommend ensuring that you only run Elasticsearch on a private network (where possible) to avoid accidentally exposing your server to the internet. Here's what our scenario will look like:



In our example, we have a private network on the `192.168.50.x` range for all

systems, with our Elasticsearch server being 192.168.50.5. Accordingly, we simply edit the configuration file (/etc/elasticsearch/elasticsearch.yml) and set the following:

```
| # Set the bind address to a specific IP (IPv4 or IPv6):  
| #  
| network.host: 192.168.50.5  
| discovery.zen.minimum_master_nodes: 1
```

We also set the minimum number of master nodes to 1, so that it knows to wait for other master nodes. In a clustered system, you ideally want three or more nodes to form a quorum to ensure the data remains valid.

So, all we need to do is start it:

```
| systemctl start elasticsearch
```

We can now test it quickly with a simple cURL call to ensure it works:

```
| curl http://192.168.50.5:9200
```

The output should look similar to this:

```
| {  
|   "name" : "Nebulon",  
|   "cluster_name" : "elasticsearch",  
|   "version" : {  
|     "number" : "5.0.0-alpha4",  
|     "build_hash" : "3f5b994",  
|     "build_date" : "2016-06-27T16:23:46.861Z",  
|     "build_snapshot" : false,  
|     "lucene_version" : "6.1.0"  
|   },  
|   "tagline" : "You Know, for Search"  
| }
```

Logstash

With Elasticsearch set up, we can now install and configure Logstash. As Logstash will be performing the log collection role, it needs to be installed on the same server as NGINX. For larger installations, you could use Filebeat (also part of the Elastic family) to act as a lightweight log forwarder and have Logstash on its own instance to parse the logs.

Like Elasticsearch, we need Java 8 or higher to run Logstash. As this demo recipe is also using Ubuntu 16.04, we can install it via the following:

```
| apt install openjdk-8-jre
```

Next, we download the latest copy of Logstash and install it:

```
| wget https://download.elastic.co/logstash/logstash/packages/debian/logstash-5.0.0-alpha
| dpkg -i logstash-5.0.0-alpha4.deb
```

Once the installation is complete, we'll now need to create the configuration file. This file will define where to look for the logs, how to parse them, and then where to send them to. While the complexity can be a bit daunting at first, it's mostly a set and forget scenario once it's working. Here's the configuration which will work for most of the configurations already outlined in this book:

```
| input {
|   file {
|     type => nginx_access
|     path => ["/var/log/nginx/*-access.log"]
|   }
| }
| filter {
|   grok {
|     match =>{"message" => "%{COMBINEDAPACHELOG}" }
|   }
| }
| output {
|   elasticsearch { hosts => ["192.168.50.5:9200"] }
|   stdout { codec => rubydebug }
| }
```

For most installations, this needs to be placed in the `/etc/logstash/conf.d/` folder with a `.conf` filename (for example, `nginx.conf`).

In the `input` section, we use the `file` plugin to monitor all the logs which have the naming pattern `*-access.log`. The `type` definition simply allows for easy filtering if your Elasticsearch server has logs from more than one source.

Then, in the `filter` section, we use the `grok` plugin to turn plain text data into a structured format. If you haven't used grok patterns before, they're similar to regular expressions and can look quite complex to start with. Because we have NGINX using the combined log format (which has already been defined as a grok pattern), the hard work has already been done for us.

Lastly, the `output` section defines where we're sending the data. Logstash can send to multiple sources (about 30 different ones), but, in this instance, we're simply sending it to our Elasticsearch server. The `hosts` setting can take an array of Elasticsearch servers, so that in a big scenario you can load balance the push of the data.

To start Logstash, we can simply call the standard init system, which for our example is `systemd`:

```
| systemctl start logstash
```

If we now load a page on any of the monitored sites, you should now have data within Elasticsearch. We can run a simple test by querying the `logstash` index and returning a record. To do this from the command line, run the following:

```
| curl http://192.168.50.5:9200/logstash-*/_search -d '{ "size": 1 }' | python -m json.t
```

I pipe the cURL command through `python` to quickly format the code; by default, it returns it in a compressed format without whitespaces. While that cuts down on the packet size, it also makes it harder to read. Here's what the output should look like:

```

{
  "_shards": {
    "failed": 0,
    "successful": 5,
    "total": 5
  },
  "hits": {
    "hits": [
      {
        "_id": "AVZEW7HGben_s6oGYLd",
        "_index": "logstash-2016.08.01",
        "_score": 1.0,
        "_source": {
          "@timestamp": "2016-08-01T04:27:54.236Z",
          "@version": "1",
          "agent": "\\Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36\\",
          "auth": "-",
          "bytes": "0",
          "clientip": "106.70.67.24",
          "host": "nginx-224-debian8",
          "httpversion": "1.1",
          "ident": "-",
          "message": "106.70.67.24 -- [01/Aug/2016:14:27:53 +1000] \\\"GET /wiki/Main_Page HTTP/1.1\\\" 304 0 \\\"-\\\" \\\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36\\\"",
          "path": "/var/log/nginx/mediawiki-access.log",
          "referrer": "\\\"-\\\"",
          "request": "/wiki/Main_Page",
          "response": "304",
          "timestamp": "01/Aug/2016:14:27:53 +1000",
          "type": "nginx_access",
          "verb": "GET"
        },
        "_type": "nginx_access"
      },
      {
        "_score": 1.0,
        "total": 11
      },
      {
        "timed_out": false,
        "took": 2
      }
    ]
  }
}

```

While the exact syntax won't make a lot of sense yet, the important bit is to note that our single log line has been parsed into separate fields. The power of this will become evident, once we completed the installation of Kibana.

Kibana

The last part of our stack is Kibana, which visualizes the Elasticsearch data. To install, we'll simply download and install the single package. As per our initial diagram, we're installing this on our Elasticsearch server:

```
wget https://download.elastic.co/kibana/kibana/kibana-5.0.0-alpha4-amd64.deb  
dpkg -i kibana-5.0.0-alpha4-amd64.deb
```

As we have Elasticsearch bound to the local IP (192.168.50.5), we need to edit the Kibana configuration (/etc/kibana/kibana.yml) and set it to look for this address:

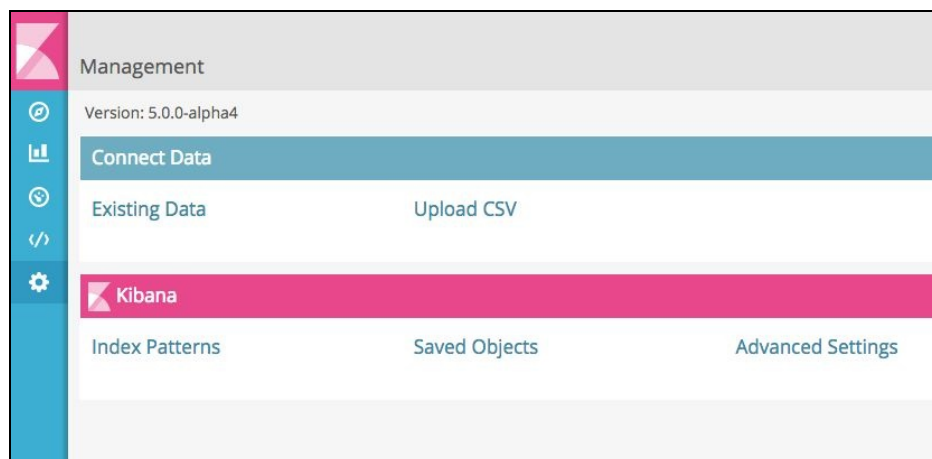
```
# The URL of the Elasticsearch instance to use for all your queries.  
elasticsearch.url: "http://192.168.50.5:9200"
```

Kibana is open to the world by default; ideally, we won't leave it in this state. Of course, one of the easiest ways to secure Kibana is by proxying it through NGINX and we'll cover this scenario in a later chapter. For now, ensure that you don't accidentally expose it to the world.

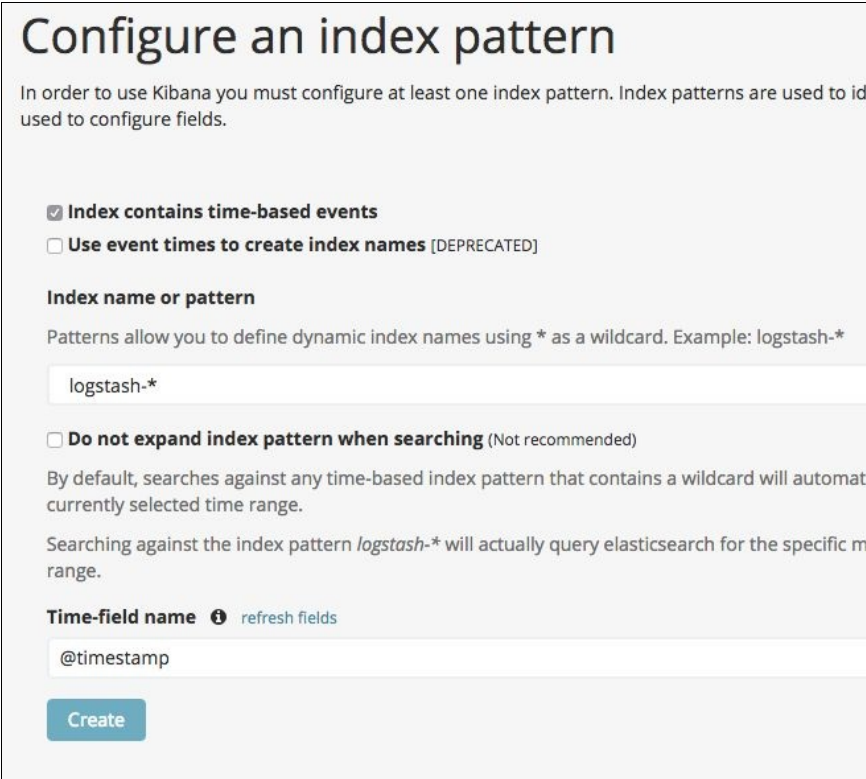
Let's start the Kibana service and get started:

```
| systemctl start kibana
```

Kibana will now be accessible via a web interface on port 5601; you should see something like this:



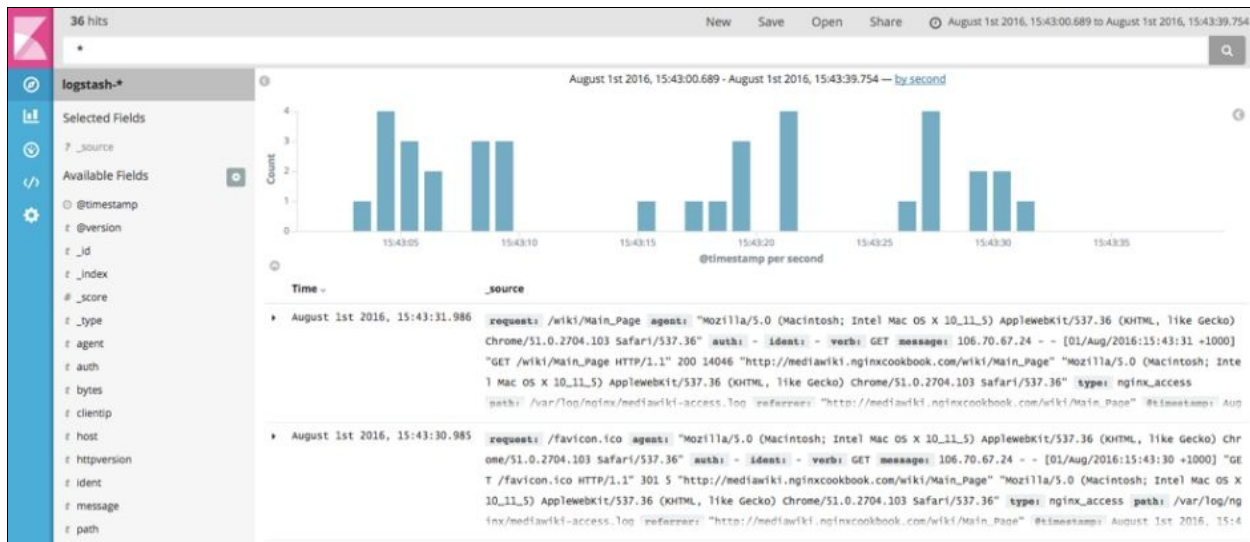
This will be a very quick crash course in the usage of Kibana, which can be complex enough to warrant a whole book in itself. To be able to visualize the data, we first need to load an index pattern into Kibana. If you click on the Discover icon (the very first at the left), it will prompt you to do so. This screen should look like the following screenshot:



The screenshot shows the 'Configure an index pattern' dialog in Kibana. The title is 'Configure an index pattern'. Below the title, there is a note: 'In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify data used to configure fields.' There are two checkboxes: 'Index contains time-based events' (checked) and 'Use event times to create index names [DEPRECATED]' (unchecked). Below these is the 'Index name or pattern' section, which includes a text input field containing 'logstash-*'. A note below the input field says: 'Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*'. There is another checkbox: 'Do not expand index pattern when searching (Not recommended)' (unchecked). Below this is a note: 'By default, searches against any time-based index pattern that contains a wildcard will automatically expand to the currently selected time range.' Another note says: 'Searching against the index pattern *logstash-** will actually query elasticsearch for the specific time range.' Below these notes is the 'Time-field name' section, which includes a text input field containing '@timestamp' and a 'refresh fields' link. At the bottom is a 'Create' button.

As our configuration is very simple, we can simply accept the default settings and hit Create. You should then see a list of fields and field types; this means that Kibana now knows how to sort, filter, and search for these.

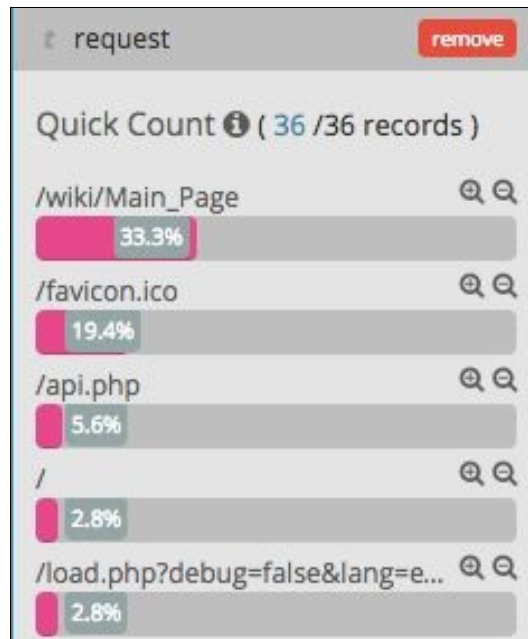
Click on the Discover icon again and it should show you the access logs in a combined bar graph and tabular format. If you don't see anything, check the time period you're searching for, which will be set in the top right. Here's what my test data looks like:



This isn't overly useful just yet, but at least we can see a quick historical view based on the hit times. To show other fields in the table, we can hover over them on the left-hand column and it should show the add button. I've added in the request, response, bytes, and httpversion to give a more detailed view like this one:

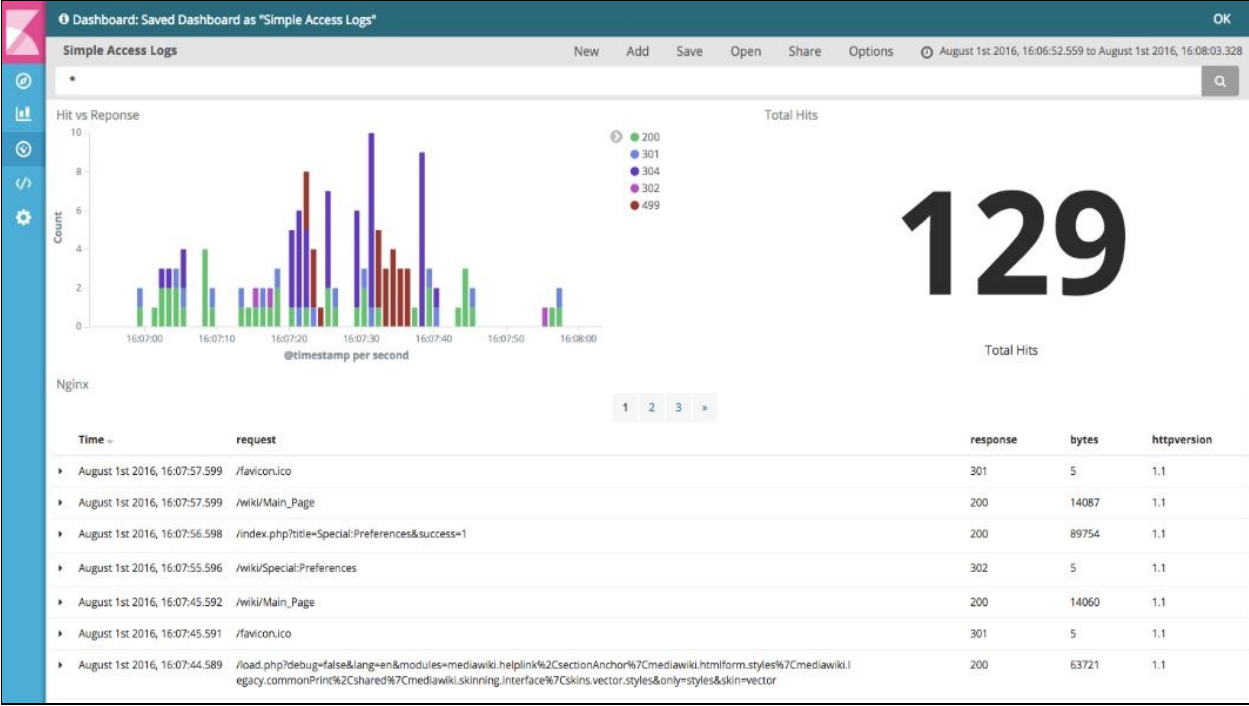
Time	request	response	bytes	httpversion
August 1st 2016, 15:43:31.986	/wiki/Main_Page	200	14046	1.1
August 1st 2016, 15:43:30.985	/favicon.ico	301	5	1.1
August 1st 2016, 15:43:30.984	/load.php?debug=false&lang=en&modules=jquery.checkboxShiftClick%2Ccookie%2CgetAttr%2ChighlightText%2CmakeCollapsible%2Cmw-jump%2CplaceHolder%2Csuggestions%7Cmediawiki.action.view.postEdit%7Cmediawiki.api%2Ccldr%2Ccookie%2CjqueryMsg%2Clanguage%2CsearchSuggest%2Ctemplate%2Cuser%7Cmediawiki.api.watch%7Cmediawiki.language.data%2Cinit%7Cmediawiki.libs.pluralRuleParser%7Cmediawiki.page.ready%7Cmediawiki.page.watch.ajax%7Csite%7Cus	200	84092	1.1
August 1st 2016, 15:43:29.982	/index.php?title=Special:UserLogin&action=submitlogin&type=login&returnto=Main+Page	302	5	1.1
August 1st 2016, 15:43:29.982	/wiki/Main_Page	200	14060	1.1
August 1st 2016, 15:43:27.979	/load.php?debug=false&lang=en&modules=jquery.checkboxShiftClick%2Ccookie%2CgetAttr%2ChighlightText%2CmakeCollapsible%2Cmw-jump%2CplaceHolder%2Csuggestions%7Cmediawiki.api%2Ccookie%2CsearchSuggest%2Cuser%7Cmediawiki.page.ready%7Cuser.default%7Cskin=vector&version=6797d90ede15	200	44840	1.1

We can start to quickly glance at the data and see if there are errors or anomalies. If we wanted to see a quick breakdown of what the requests have been, we could click on the field in the left-hand column and it would display the top five results within the search. For example, here's our demo output:



At a quick glance, we know what the most popular requests have been. We can also filter based on these values, by clicking on the magnifying glasses at the right of the dialog. The positive icon will ensure only those which match this request are shown and the negative icon will exclude anything which matches this request.

This is just the beginning too, as you'll see on the left-hand side, there's the ability to create visualizations, which can then be placed in a full dashboard. You can add any number of line graphs, bar graphs, and similar items based on predetermined filters and searches. Of course, you can have multiple dashboards, so that you can customize them for multiple different scenarios. As one example, here's a quick dashboard which was created in about five minutes:



See also

The Elastic Stack documentation: <https://www.elastic.co/guide/index.html>

Rewrites

In this chapter, we will cover the following recipes:

- Redirecting non-www to www-based sites
- Redirecting to a new domain
- Blocking malicious user agents
- Redirecting all calls to HTTPS to secure your site
- Redirecting pages and directories
- Redirecting 404 errors through a search page

Introduction

The ability to rewrite URLs gives us a number of different abilities within NGINX. First and foremost, we can display neat URLs to the end user when we have a dynamically driven system such as WordPress. For example, instead of trying to remember `index.php?page_id=124`, we can simply use `/about/`.

Secondly, when we move content, change platforms, or update a system, we want to make sure all the old URLs still work. This is especially important for **Search Engine Optimization (SEO)**, as well as for anyone who has your old page bookmarked.

In this chapter, we'll run through a number of recipes describing how to activate all of the common rewrites and also describing a few of these rewrites to provide great flexibility.

Redirecting non-www to www-based sites

The debate about using a non-www prefixed URL versus including the www still rages on in the SEO world, but the key thing is to ensure there is consistency.

This means that if you choose to have the `www` prefix, then you'll want to ensure all URL calls are to this pattern. Especially since there are SEO implications involved with any URL change, attention to detail is critical here. For this recipe, we'll redirect all non-www to www-based calls, but the reverse is simply a matter of slightly modifying the rewrite rule.

How to do it...

If you're used to the Apache way of doing things, this may look a little different. In the NGINX world, *if is evil* and needs to be avoided at all costs. Instead, we set one `server` block up for the non-www site and set it to redirect and a separate `server` block for the www-based site. Here's an example code:

```
server {
    listen      80;
    server_name redirect.nginxcookbook.com;
    return      301
    http://www.redirect.nginxcookbook.com$request_uri;
}

server {
    listen      80;
    server_name www.redirect.nginxcookbook.com;
    location / {
        root    /usr/share/nginx/html;
        index   index.html index.htm;
    }
}
```

Using the Chrome DevTools (or the equivalent), we can see the following sequence:

Name	Met...	Status	Protocol	Type	Initiator
 redirect.nginxcookbook.com	GET	301	http/1.1	text/html	Other
 www.redirect.nginxcookbook.com	GET	200	http/1.1	document	http://redirect.nginxcookbook.com/

The original request to `redirect.nginxcookbook.com` sends a `301` to immediately load `www.redirect.nginxcookbook.com`. The browser then makes a request for this new URL, which, in the preceding example, responded with a HTTP 200.

How it works...

In this rewrite, we use a 301 HTTP call, which means that the URL has moved permanently. This indicates to any upstream system (be it a web crawler or proxy) that the change is final and not to look for the old URL. From an SEO perspective, this is critical, as any links or rankings applied to the old link should then be transferred to the new link.

The `$request_uri` variable passes the full URI, including arguments. This means that scenarios such as subdirectories (for example, `/about/`) as well as any arguments (for example, `id=123`) are redirected. This ensures that all bookmarks and direct links will remain working seamlessly after the change.

See also

- Moz redirection: <https://moz.com/learn/seo/redirection>
- NGINX redirect blog: <https://www.nginx.com/blog/creating-nginx-rewrite-rules/>

Redirecting to a new domain

If a product or company decides to rebrand and rename, then one key part of this is to redirect the old domain to the new one. Even the top names such as Google (who started as BackRub) and eBay (who started as AuctionWeb) have had to go through this process, so it's more common than people realize.

In order to maintain all previous links and functionalities, ensuring all URLs are redirected to the new domain is critical. Thankfully, all that's required is a simple redirect with a rewrite to ensure that the full URL and all arguments are correctly sent to the new domain.

```
server {
```

```
    listen 80; server_name oldsite.nginxcookbook.com; return 301  
    http://newsite.nginxcookbook.com$request_uri; }
```

```
server {
```

```
    listen 80; server_name newsite.nginxcookbook.com; location / {  
        root /usr/share/nginx/html; index index.html index.htm; }
```

```
}
```

How it works...

Like our non-www redirect, this recipe works in the same fashion. In fact, a redirect for a subdomain isn't functionally any different from a completely different domain altogether.

The return rule works just the same, where calls to subdirectories and all arguments are passed to the new domain without breaking their functionality. For example, using our preceding code, if we made a call to `http://oldsite.nginxcookbook.com/pages/?item=1951`, it would send a 301 redirect to `http://newsite.nginxcookbook.com/pages/?item=1951`.

There's more...

While redirecting the domain fixes one part of the problem, you may need to make code changes as well. CMSes such as WordPress also need to have both the site URL and internal links updated as well. Make sure you consult the documentation for your respective application to ensure you take the necessary steps to migrate internal calls.

Blocking malicious user agents

As the saying goes: *"Just because you're not paranoid doesn't mean they aren't after you."*

—Joseph Heller.

In the web world, hackers are continually scanning your servers and sites regardless of who you are and what sort of site you have. In 99 percent of all instances, they don't care who you are or what your site is, but only see how easily they can manipulate it to do what they want.

Even if you're 100 percent sure that your website is secure, you can save a lot of CPU cycles and potential **Denial of Service (DOS)** attacks by blocking these agents.

```

server {

    listen 80; server_name badbots.nginxcookbook.com;

    if ($http_user_agent ~ <br/>
(Baiduspider|Yandex|DirBuster|libwww|"")) {

        return 403; }

    location / {

        root /usr/share/nginx/html; index index.html index.htm; }

}

```

```

106.74.67.24 - - [04/Sep/2016:22:24:03 +1000] "GET / HTTP/1.1"
403 571 "-" "libwww" "-"

```

```

106.74.67.24 - - [04/Sep/2016:22:24:03 +1000] "GET / HTTP/1.1"
403 571 "-" "libwww" "-"

```

```

106.74.67.24 - - [04/Sep/2016:22:24:03 +1000] "GET / HTTP/1.1"
403 571 "-" "libwww" "-"

```

```

106.74.67.24 - - [04/Sep/2016:22:24:04 +1000] "GET / HTTP/1.1"
403 571 "-" "libwww" "-"

```

```

106.74.67.24 - - [04/Sep/2016:22:24:04 +1000] "GET / HTTP/1.1"
403 571 "-" "libwww" "-"

```

How it works...

Before our `location` block, we add an `if` statement to compare against the user agent. This ensures it's evaluated for every call. If your intent is to drop the CPU load by blocking malicious requests, it can be a good compromise.

```
| if ($http_user_agent ~ (Baiduspider|Yandex|DirBuster|libwww|"")) {  
|     return 403;  
| }
```

The tilde (`~`) performs a case-sensitive match against the user agent (`$http_user_agent`), and allows for a partial match. This means that if the user agent string contains `Yandex1.0` OR `Yandex2.0`, both will still match the rule.

The list of blocked agents (`Baiduspider|Yandex|DirBuster|libwww|""`), uses the pipe (`|`) as an OR so that any of the strings can be matched. The double quotes at the end are there to block any system which doesn't report any user agent.

If you need complexity and more dynamic protection, then a proper **Web Application Firewall (WAF)** may be a better fit. A WAF has the ability to block other threats, such as SQL injection and **Cross-site Scripting (XSS)** as well.

```
if ($http_user_agent ~ "WordPress") {  
    return 403; }
```

With this rule in place, the site remained operational and the only negative issue was slightly increased traffic. The server was easily able to cope with the normal traffic and the rise in CPU usage was only minimal. Being able to turn a site which was completely offline back into an operational site within five minutes was a great outcome.

Redirecting all calls to HTTPS to secure your site

Back in [Chapter 4](#), *All About SSLs*, we covered installing and configuring an SSL certificate with NGINX. However, one thing we didn't cover was, ensuring that all calls to your site or application are always encrypted. This is especially critical if you're handling private data or payment information, which could be mandated by law to ensure the transmissions of the data is encrypted. Thankfully, enforcing HTTPS is simple to do.

```
server {  
  
    listen 80;  
  
    server_name ssl.nginxcookbook.com; return 301  
https://ssl.nginxcookbook.com$request_uri; }
```

```
server {  
  
    listen 443 ssl; server_name ssl.nginxcookbook.com; ssl_certificate  
/etc/ssl/public.pem; ssl_certificate_key /etc/ssl/private.key;  
ssl_protocols TLSv1 TLSv1.1 TLSv1.2; ssl_ciphers  
HIGH:!aNULL:!MD5;  
  
    access_log /var/log/nginx/ssl-access.log combined;  
  
    location / {  
  
        root /var/www; index index.html index.htm; }  
  
}
```

Here, we have our standard server block listening on port 80 and then redirecting all calls back to our calls to the HTTPS version. With the whole site redirected, even if there's a mistaken link to the HTTP version of the site, it will be automatically redirected.

```
server {  
  
    listen 80 default_server; server_name _; return 301  
    https://$server_name$request_uri; }
```

Set `default_server` to the `server` block of any undefined server. The underscore (`_`) is simply a placeholder in NGINX; because we've set this `server` block to be the default, it will process all non-matching requests anyway. The underscore simply makes it easy to view.

See also

The NGINX `server_name` documentation: http://nginx.org/en/docs/http/server_names.html

Redirecting pages and directories

One of the most common uses of rewrites is to automatically redirect a URL from an old platform to your new site. If you've ever changed your platform (for example, moved from a proprietary system to WordPress or similar), then you'll have a large amount of existing links which will already be bookmarked and ranked within Google.

Like our [404](#) recipe, many modern CMSes can do this, but most can't do it with the efficiency that NGINX can. Anything which can reduce the server load on a production system is always a good thing, especially if you have a very busy site.

How to do it...

The following are a number of quick references to redirect your page or directory.

Single page redirect

If you've moved a page or have a new CMS, you can redirect each page using the following:

```
| rewrite ^/originalpage/$ /newpage/ redirect;
```

This simply has to be within your `server` block directive. It's important to note that the syntax needs to be an exact match. In the preceding example, if you called the URL without the trailing slash, it would fail to redirect. We can account for both these scenarios by modifying the `rewrite` directive quickly:

```
| rewrite ^/originalpage/?$ /newpage/ redirect;
```

As explained in detail later, this means that the trailing slash is now optional, so that both call to `/originalpage` and `/originalpage/` will now both redirect.

Full directory redirect

If you want everything within a whole directory to redirect, this can also be easily covered with one line:

```
| rewrite ^/oldproduct/?(.*)$ /newproduct/$1 redirect;
```

Again, this needs to go within your existing `server` block directive. While similar to the single page, there are a few key differences. First, we add a subexpression `(.*)`, which we then pass to the new URL as a variable `($1)`.

For example, if we call the `/oldproduct/information` URL, it will redirect to `/newproduct/information`. This will match anything in the URL, so if you have parameters in the call (for example, `id=1`), these will also be passed to the new URL.

How it works...

The rewrite rules use **Perl Compatible Regular Expressions (PCRE)**, which is a pattern matching system for textual content. While the more complex scenarios can be quite complex to read, you can quickly grasp the basics by understanding the basics of the syntax. Here's a reference table:

Syntax	Meaning
<code>^</code>	The start of the string to match
<code>\$</code>	The end of the string to match
<code>.</code>	Any character (except newline)
<code>\d</code>	Any digit
<code>\D</code>	Any non-digit
<code>\w</code>	Word, meaning letters, digits, and underscore (<code>_</code>)
<code>*</code>	Zero or more of the previous expression
<code>+</code>	One or more of the previous expression

?	Zero or one of the previous expression
()	Group the expression to allow the subexpression to be used separately

These are just the basics, but it does allow a very rapid understanding of the more simplistic rules. The following table has a number of quick examples of a match against the `/oldproduct/show.php?id=1` string:

Rule	Meaning	Match
<code>^.*\$</code>	This matches everything on a line	<code>/oldproduct/show.php?id=1</code>
<code>^/oldproduct/\$</code>	This looks for a string which must be exactly <code>/oldproduct/</code>	No match
<code>^/oldproduct/</code>	This looks for a string which must start with <code>/oldproduct/</code> but only matches that phrase	<code>/oldproduct/</code>
<code>^/oldproduct.*</code>	This looks for a string which must start with <code>/oldproduct/</code> and also matches everything after that	<code>/oldproduct/show.php?id=1</code>
<code>\w+\.php</code>	This looks for any letters, numbers, or underscores and match at least 1 ending in <code>.php</code>	<code>show.php</code>

Unless you do it day in and day out, don't expect to have your regular expression

work perfectly the first time. There are a number of handy tools online (such as regex101.com), which allows you to test your expressions and give a detailed feedback. For complex scenarios, these tools are critical and allow detailed testing before adding the rules to a live server.

See also

The regular expression test tool: <https://regex101.com/>

Redirecting 404 errors through a search page

If you have moved URLs or have an incorrect link to your site, it will generally generate a 404 Not Found error to the browser. While you can also style this page to make it match the site, the end result is generally the same. If a user hits this link, either way, they won't find the content they were originally searching for.

With a small amount of trickery, we can instead turn the 404 into search parameters to push through to your site. This way, you can either match the page or item the user was looking for or at least display the best guesses for them.

While most modern CMSes either have this functionality built in or have a plugin to handle it for you, if you're stuck with an older system or custom coded site, then you can quickly add this functionality.

```
server {  
  
    listen 80;  
  
    server_name demosite.nginxcookbook.com;  
  
    access_log /var/log/nginx/demosite.access.log combined; index  
index.php;  
  
    root /var/www/vhosts/demosite;  
  
    error_page 404 @404page;  
  
    location @404page {  
  
        try_files $uri $uri/ /search.php?uri=$uri&$args; }  
  
  
    location ~ \.php$ {  
  
        fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index  
index.php; fastcgi_param SCRIPT_FILENAME  
$document_root$fastcgi_script_name; include fastcgi_params;  
  
        try_files $uri $uri/ /search.php?uri=$uri&$args; }  
  
}
```

In this server block, we've assumed that the site uses PHP, but static files with a search file just for the 404 errors will also work.

How it works...

For our 404 errors, we override the default NGINX 404 page to a custom location (`error_page 404 @404page;`). The reason we need to use a custom location is so that we can rewrite the URI and arguments.

Within our `@404page` location, we rewrite the URI and arguments to the search page (`/search.php?uri=$uri&$args`), so that they can be used as variables. For example, if we make a call to `/oldpage/index.asp?id=44`, it will send this to `/search.php` as the following variables:

- `[uri] => /oldpage/index.asp`
- `[id] => 44`

Lastly, we also include a `try_files` call to ensure that any missing PHP files are also rewritten to the `search.php` file.

The `search.php` file could include the ability to have a lookup table or it could incorporate a database search to do a fuzzy search and either automatically redirect or simply present options to the user.

Reverse Proxy

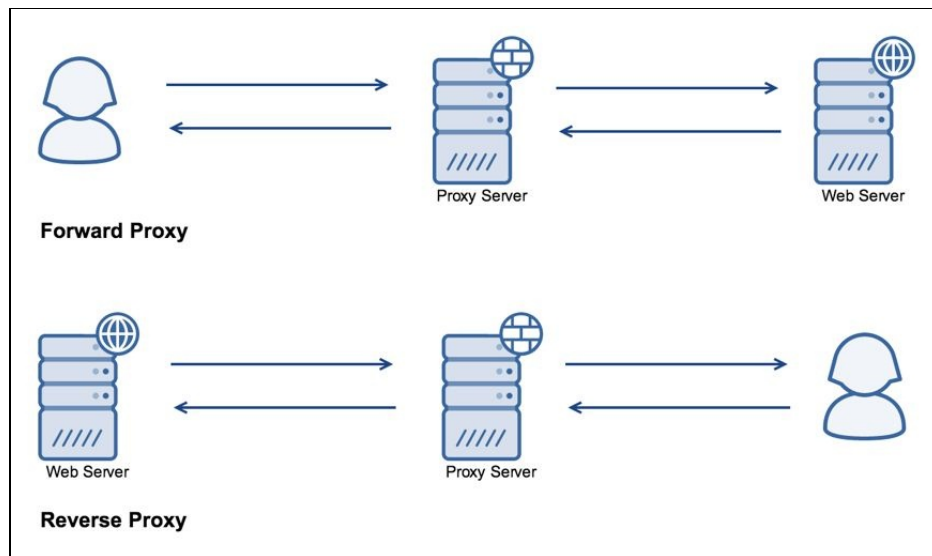
In this chapter, we will cover the following recipes:

- Configuring NGINX as a simple reverse proxy
- Content caching with NGINX
- Monitoring cache status
- Microcaching
- Serving from cache when your backend is down
- SSL termination proxy
- Rate limiting

Introduction

One of the most powerful features of NGINX is its ability to act as a reverse proxy. As opposed to a forward proxy, which sits between the client and the internet, a reverse proxy sits between a server and the internet.

Here's a visual representation:



A reverse proxy can provide a multitude of features. It can load balance requests, cache content, rate limit, provide an interface to a **Web Application Firewall (WAF)**, and lots more. Basically, you can greatly increase the number of features available to your system by running it through an advanced reverse proxy.

Configuring NGINX as a simple reverse proxy

In this recipe, we'll look at the basic configuration of a simple reverse proxy scenario. If you've read the first few chapters, then this is how NGINX was configured in front of PHP-FPM and similar anyway.

Getting ready

Before you can configure NGINX, you'll first need to ensure your application is listening on a different port than port 80, and ideally on the `loopback` interface, to ensure it's properly protected from direct access.

How to do it...

Here's our `server` block directive to proxy all requests through to port `8000` on the localhost:

```
server {  
    listen      80;  
    server_name proxy.nginxcookbook.com;  
    access_log  /var/log/nginx/proxy-access.log combined;  
  
    location / {  
        proxy_pass http://127.0.0.1:8000;  
    }  
}
```

How it works...

For all requests, the `location` block directive is set to proxy all connections to a specified address (`127.0.0.1:8000` in this instance). With the basic settings, NGINX doesn't manipulate any of the data, nor does it cache or load balance. Testing with a basic proxy method is always a good step before moving to a complicated configuration to ensure that your application or program will work as expected.

To get around the fact that you don't want private information cached (and therefore potential information leakage), NGINX will check for both cookies and cache headers. For example, when you log in to a WordPress system, this login will return a `Set-Cookie` header and NGINX will therefore exclude this from being cached.

```
server {  
  
    listen 80; server_name proxy.nginxcookbook.com; access_log  
    /var/log/nginx/proxy-access.log combined;  
  
    location / {  
  
        proxy_pass http://127.0.0.1:8000; proxy_set_header X-Forwarded-  
        For <br/> $proxy_add_x_forwarded_for; proxy_set_header X-Real-IP  
        $remote_addr; proxy_set_header Host $host; }  
  
    }
```

The X-Forwarded-For header shows the full chain of servers which have proxied the packet, which could be multiple forward proxies. This is why we also have X-Real-IP, so that we ensure we have the real IP address of the client.

To ensure that the upstream hostname is sent through, we set the Host header field. This allows the upstream server to be name-based and can allow multiple hosts (that is, multiple websites) to be proxied under one configuration or one server.

See also

- Official NGINX proxy help: http://nginx.org/en/docs/http/nginx_http_proxy_module.html
- The NGINX proxy guide: <https://www.nginx.com/resources/admin-guide/reverse-proxy/>

Content caching with NGINX

In addition to simply proxying the data through, we can use NGINX to cache the proxied content. By doing this, we can reduce the amount of calls to your backend service, assuming that the calls are able to be cached.

Getting ready

As the caching is part of the standard NGINX platform, no additional prerequisites are required.

```
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=cache:2m
```

```
server {
```

```
    listen 80; server_name cached.nginxcookbook.com; access_log  
/var/log/nginx/cache-access.log combined;
```

```
    location / {
```

```
        proxy_pass http://localhost:8080; proxy_cache cache;  
        proxy_set_header X-Real-IP $remote_addr; proxy_set_header X-  
Forwarded-For $remote_addr; proxy_set_header Host $host; }
```

```
}
```

How it works...

Firstly, the first parameter to the `proxy_cache_path` directive is the location of the cache files. You can choose any directory which makes sense to your server structure, but ensure that the NGINX user on your server has write access.

The `levels` parameter specifies how the cache is written to the system. In our recipe, we have specified `1:2`. This means that the files are stored in a two-level hierarchy. The reason this is configurable is due to potential slowdowns when there are thousands of files within a single directory. Having two levels is a good way to ensure this never becomes an issue.

The third parameter, `keys_zone`, sets aside memory to store metadata about the cached content. Rather than a potentially expensive (system resource wise) call to see whether the file exists or not, NGINX will map the file and use in-memory metadata for tracking. In our recipe, we have allocated 2 MB and this should be sufficient for up to 16,000 records.

There's more...

While writing the cache to disk may seem counter-intuitive from a performance perspective, you need to take into account that the Linux kernel will cache file access to memory. With enough free memory, the file will be read once and then each subsequent calls will be direct from RAM. While this may take more memory than a standard configuration, a typical website can be as little as 64 MB in total, which is trivial by modern standards.

Having the cache disk-based means that it's also persistent between reboots or restarts of NGINX. One of the biggest issues with the cold start of a server is the load on the system until it has had a chance to warm the cache. If you need to ensure that the loading of any cache file from disk is as fast as possible, I'd recommend ensuring that the cache is stored on a high-speed **Solid State Drive (SSD)**.

See also

The NGINX caching guide: <https://www.nginx.com/blog/nginx-caching-guide/>

Monitoring cache status

When developing complex sites or rapidly changing content, one key aspect is to monitor where the content was served from. Essentially, we need to know whether we hit the cache or whether it was a miss.

This helps us ensure that, if there are issues, or on seeing incorrect content, we know where to look. It can also be used to ensure the caching is working on pages where it's expected and being bypassed for areas where it shouldn't be.

Getting ready

As the caching is part of the standard NGINX platform, no additional prerequisites are required.

How to do it...

One of the easiest ways is to simply add an additional header. To do this, we add the additional directive to our existing proxy configuration within the `location` block directive:

```
| proxy_set_header X-Cache-Status $upstream_cache_status;
```

httpstat http://proxy.nginxcookbook.com/

HTTP/1.1 200 OK

Server: nginx/1.11.2

Date: Sun, 09 Oct 2016 12:18:54 GMT

Content-Type: text/html; charset=UTF-8

Transfer-Encoding: chunked Connection: keep-alive Vary: Accept-Encoding X-Powered-By: PHP/7.0.12

X-Cache-Status: HIT

We can see by the x-cache-status header that the request was a hit, meaning it was served from the cache not the backend. Other than the basic hit and miss, there are also a number of other statuses which could be returned:

Status	Meaning
HIT	The request was a hit and therefore served from cache
MISS	The request wasn't found in cache and therefore had to be requested from the backend server
BYPASS	The request was served from the backend server, as NGINX was explicitly told to bypass the cache for the request
EXPIRED	The request had expired in cache, so NGINX had to get a new copy from the backend server
STALE	NGINX couldn't talk to the backend server, but instead has been told to serve stale content
UPDATING	NGINX is currently awaiting an updated copy from the backend

server, but has also been told to serve stale content in the interim

REVALUATED This relies on the use of `proxy_cache_revalidate` being enabled and checks the cache control headers from the backend server to determine if the content has expired

See also

- NGINX HTTP upstream variables: http://nginx.org/en/docs/http/nginx_http_upstream_module.html#variables
- The `httpstat` utility: <https://github.com/reorx/httpstat>

Microcaching

Caching is a great way to speed up performance, but some situations mean that you will be either continually invalidating the content (which means you'll need more server resources) or serving stale content. Neither scenario is ideal, but there's an easy way to get a good compromise between performance and functionality.

With microcaching, you can set the timeout to be as low as one second. While this may not sound like a lot, if you're running a popular site, then trying to dynamically serve 50+ requests per second can easily bring your server to its knees. Instead, microcaching will ensure that the majority of your requests (that is, 49 out of the 50) are served direct from cache, yet will only be 1 second old.

Getting ready

As the caching is part of the standard NGINX platform, no additional prerequisites are required.

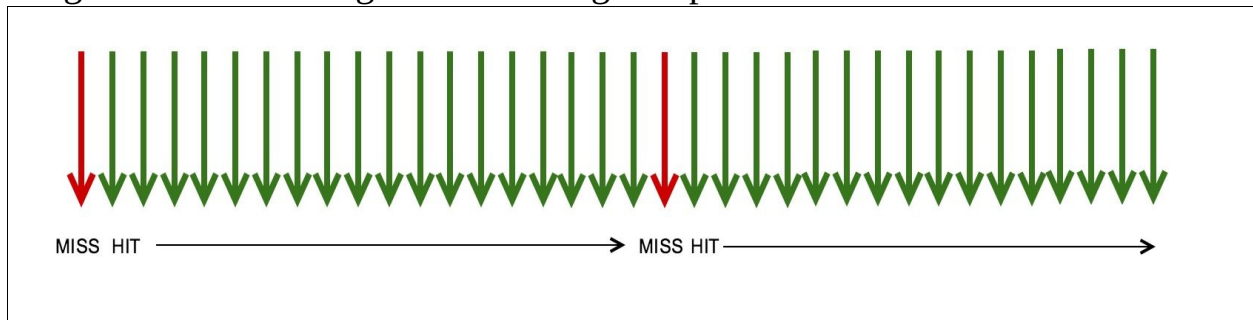
How to do it...

To take advantage of microcaching, we expand on our previous recipe to reduce the timeout of the cache.

```
server {  
    listen      80;  
    server_name micro.nginxcookbook.com;  
  
    access_log /var/log/nginx/micro-access.log combined;  
  
    location / {  
        proxy_pass http://127.0.0.1:8080;  
        proxy_cache micro;  
        proxy_cache_valid 200 10s;  
        proxy_set_header X-Real-IP $remote_addr;  
        proxy_set_header X-Forwarded-For $remote_addr;  
        proxy_set_header Host $host;  
    }  
}
```

When you first look at it, 1 second timeouts seem like they won't provide any help. For a busy site, the reduction of requests which have to hit the backend server can be significantly dropped. It also means that your burst ability is greatly increased, allowing your site to handle a spike in traffic without issue.

As shown in the following diagram, the higher the **Requests Per Second (RPS)** the greater the advantage microcaching will provide:



How it works...

We set the `proxy_cache_valid` directive to cache the `200` responses and set them to be valid for 1 second (`1s`).

The validation value can be as low as your minimum content refresh. If you need changes to be live instantly, a 1 second timeout for the validation can be used. If you have less frequent changes, then setting 10-20 seconds can also be acceptable for many sites.

HTTP response codes of `200` represent an OK response, meaning it was a successful response from the server. We could also cache `404` requests (Not Found) as well, especially knowing that some of these can be quite resource intensive if they involve database searches.

Serving from cache when your backend is down

While we don't want to see a scenario where your backend server is down, expecting to maintain 100 percent uptime simply isn't realistic. Whether it's unexpected or planned upgrades, having the ability to still serve content is a great feature.

With NGINX, we can tell it to serve stale cache data when it can't reach your backend server. Having a page which is slightly out-of-date is (in most scenarios) a far better outcome than sending the client a 502 HTTP error (Bad Gateway).

Getting ready

As the caching is part of the standard NGINX platform, no additional prerequisites are required.

```
server {  
  
    listen 80; server_name proxystale.nginxcookbook.com; access_log  
/var/log/nginx/proxy-access.log combined;  
  
    location / {  
  
        proxy_pass http://127.0.0.1:8000; proxy_set_header X-Real-IP  
$remote_addr; proxy_set_header X-Forwarded-For $remote_addr;  
proxy_set_header Host $host; proxy_cache_use_stale error timeout  
http_500 http_502 <br/> http_503 http_504; }  
  
}
```

How it works...

With the `proxy_cache_use_stale` directive, we specify which cases should use a stale copy of the cache. For this recipe, we've specified that when we have an error, timeout, `500` (Internal Server Error), `502` (Bad Gateway), `503` (Service Unavailable), and `504` (Gateway Timeout) errors from the backend server that the stale copy can be used.

If any of these scenarios trigger, we take the less abrupt option of serving the content. Especially, if you've got short cache times (such as microcaching), the users of the site won't even notice the difference.

There's more...

One thing you don't want to do is queue thousands of requests as your backend system comes online. With a busy site, it's easy to overwhelm your system the moment it becomes available again. Especially since a big update or restart usually means local object caches within the backend are also cold, care needs to be taken when bringing it back online.

One great feature NGINX has is to lock the cache to ensure only one request per unique key is sent through. Here's an updated configuration to use this lock:

```
location / {
    proxy_pass http://www.verifiedparts.com:80;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_set_header X-Cached $upstream_cache_status;
    proxy_set_header Host www.verifiedparts.com;
    proxy_cache_use_stale error updating timeout http_500 http_502
    http_503 http_504;
    proxy_cache_lock on;
    proxy_cache_lock_timeout 60s;
}
```

The `proxy_cache_lock` directive ensures that only one request (if there's a cache `MISS`) is sent through to the backend/upstream server. All other requests are either served from the cache (and stale if using this recipe) until the timeout directive (`proxy_cache_lock_timeout`) is triggered, and if the cache status is still a `MISS`, then it will try again. The timeout value needs to be sufficient to allow the backend to be ready to serve pages; for some .NET-based or Java systems, this could be as high as 120 seconds.

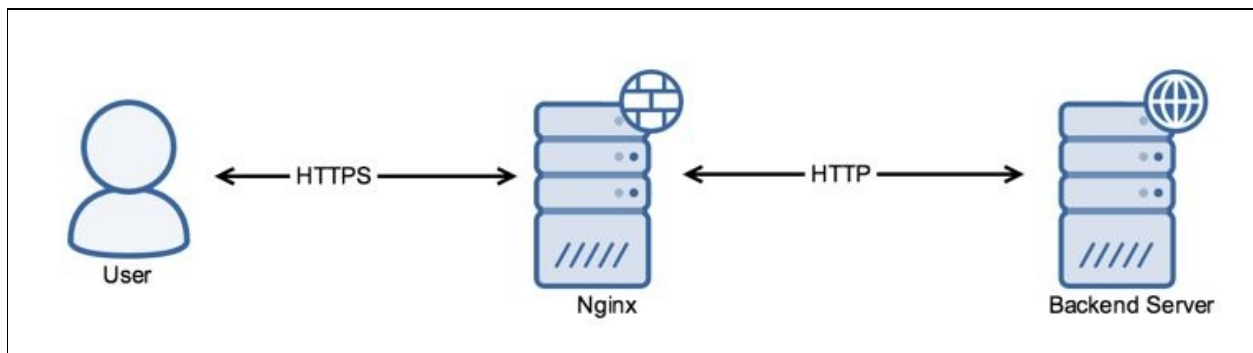
This combination greatly lowers the peak impact on the backend after a cold start and helps to avoid overwhelming the system. By ensuring only one request per URI can be directed at the backend, we help ensure it has time to properly process requests as the cache warms again.

See also

- The proxy module: http://nginx.org/en/docs/http/nginx_http_proxy_module.html#proxy_cache_use_stale
- The cache lock documentation: http://nginx.org/en/docs/http/nginx_http_proxy_module.html#proxy_cache_lock

SSL termination proxy

One of the first use cases that I tried NGINX out for was simply as an SSL termination proxy. If you have an application which can't directly produce HTTPS (encrypted) output, you can use NGINX as a proxy to do this. Content is served from your backend in plain text, then the connection between NGINX and the browser is encrypted. To help explain, here's a diagram covering the scenario:



The advantage is that you also get to make use of all the other NGINX feature sets too, especially when it comes to caching. In fact, if you've used the Cloudflare service to achieve a similar outcome, then you may be surprised to know that it's NGINX-based as well.

Getting ready

This recipe involves the use of SSL certificates. If you haven't currently generated any for your deployment, see [Chapter 4](#), *All About SSLs*, for hints and tips.

```
server {  
  
    listen 443 ssl; server_name ssl.nginxcookbook.com; ssl_certificate  
/etc/ssl/public.pem; ssl_certificate_key /etc/ssl/private.key;  
ssl_protocols TLSv1 TLSv1.1 TLSv1.2; ssl_ciphers  
HIGH:!aNULL:!MD5;  
  
    access_log /var/log/nginx/ssl-access.log combined;  
  
    location / {  
  
        proxy_pass http://localhost:8000; proxy_set_header X-Real-IP  
$remote_addr; proxy_set_header X-Forwarded-For $remote_addr;  
proxy_set_header Host $host; }  
  
}
```

The following are some useful tips you should keep in mind:

- One thing to note is that most SSL certificates are only valid for a single domain, unless they're a wildcard or **Subject Alternative Name (SAN)**. If you're intending to use NGINX as an SSL terminator to multiple hosts, you'll need to have a server block or a SAN certificate mapped for each host.
- Be careful with internal redirects within your application, especially if you tell it to enforce HTTPS. When using NGINX for SSL termination, this needs to be done at the NGINX level to avoid redirect loops.

See also

The NGINX SSL termination guide: <https://www.nginx.com/resources/admin-guide/nginx-ssl-termination/>

Rate limiting

If you have an application or site where there's a login or you want to ensure fair use between different clients, rate limiting can help to help protect your system from being overloaded.

By limiting the number of requests (done per IP with NGINX), we lower the peak resource usage of the system, as well as limit the effectiveness of attacks which are attempting to brute force your authentication system.

```
limit_req_zone $binary_remote_addr zone=basiclemit:10m rate=10r/s;
```

```
server {
```

```
    listen 80;
```

```
    server_name limit.nginxcookbook.com; access_log  
/var/log/nginx/limit-access.log combined;
```

```
    location / {
```

```
        limit_req zone=basiclemit burst=5; proxy_pass  
http://127.0.0.1:8000; proxy_set_header X-Forwarded-For <br/>  
$proxy_add_x_forwarded_for; proxy_set_header X-Real-IP  
$remote_addr; proxy_set_header Host $host; }
```

```
    }
```

```
<strong>ab -c 1 -n 200 http://limit.nginxcookbook.com/</strong>
```

Concurrency Level: 1

Time taken for tests: 20.048 seconds Complete requests: 200

Failed requests: 0

Total transferred: 5535400 bytes HTML transferred: 5464000 bytes
Requests per second: 9.98 [#/sec] (mean) Time per request: 100.240
[ms] (mean) Time per request: 100.240 [ms] (mean, across all
concurrent
 requests) Transfer rate: 269.64 [Kbytes/sec] received

```
<strong>ab -c 4 -n 200 http://limit.nginxcookbook.com/</strong>
```

Concurrency Level: 4

Time taken for tests: 20.012 seconds Complete requests: 200

Failed requests: 0

Total transferred: 5535400 bytes HTML transferred: 5464000 bytes
Requests per second: 9.99 [#/sec] (mean) Time per request: 400.240
[ms] (mean) Time per request: 100.060 [ms] (mean, across all
concurrent
 requests) Transfer rate: 270.12 [Kbytes/sec] received

Even with the increased request rate, we still received responses at a
rate of 10 requests per second.

How it works...

There are a number of aspects to this recipe to consider. The first is the `limit_req_zone` directive within the main NGINX configuration file. We can create multiple zones for tracking and base them on different tracking parameters. In our recipe, we used `$binary_remote_addr` to track the remote IP address. The second parameter is then the name of the zone and the memory to allocate. We called the `basiclimit` zone and allocated 10 MB to it, which is sufficient to track up to 160,000 IP addresses. The third parameter is the rate, which we set to 10 requests per second (`10r/s`).



If you need to have different rate limits for different sections (for example, a lower rate limit for an admin login area), you can define multiple zones with different names.

To utilize the zone, we then added it to one of the existing location directives using `limit_req`. For our recipe, we specified the zone we created (`basiclimit`) and also gave it a burst ability of 5. This burst allows for a small buffer over the specified limit before errors are returned and helps to smooth out responses.

limit_req zone=basiclimit burst=5 nodelay;

Concurrency Level: 1

Time taken for tests: 4.306 seconds Complete requests: 200

Failed requests: 152

(Connect: 0, Receive: 0, Length: 152, Exceptions: 0) Non-2xx responses: 152

Total transferred: 1387016 bytes HTML transferred: 1343736 bytes
Requests per second: 46.45 [#/sec] (mean) Time per request: 21.529 [ms] (mean)
Time per request: 21.529 [ms] (mean, across all concurrent requests) Transfer rate: 314.58 [Kbytes/sec] received

Not all our requests returned with a status of 200, and instead any requests over the limit immediately received a 503. This is why our benchmark only shows 46 successful requests per second, as 152 of these were 503 errors.

See also

The NGINX `ngx_http_limit_req_module`: http://nginx.org/en/docs/http/ngx_http_limit_req_module.html.

Load Balancing

In this chapter, we will cover the following:

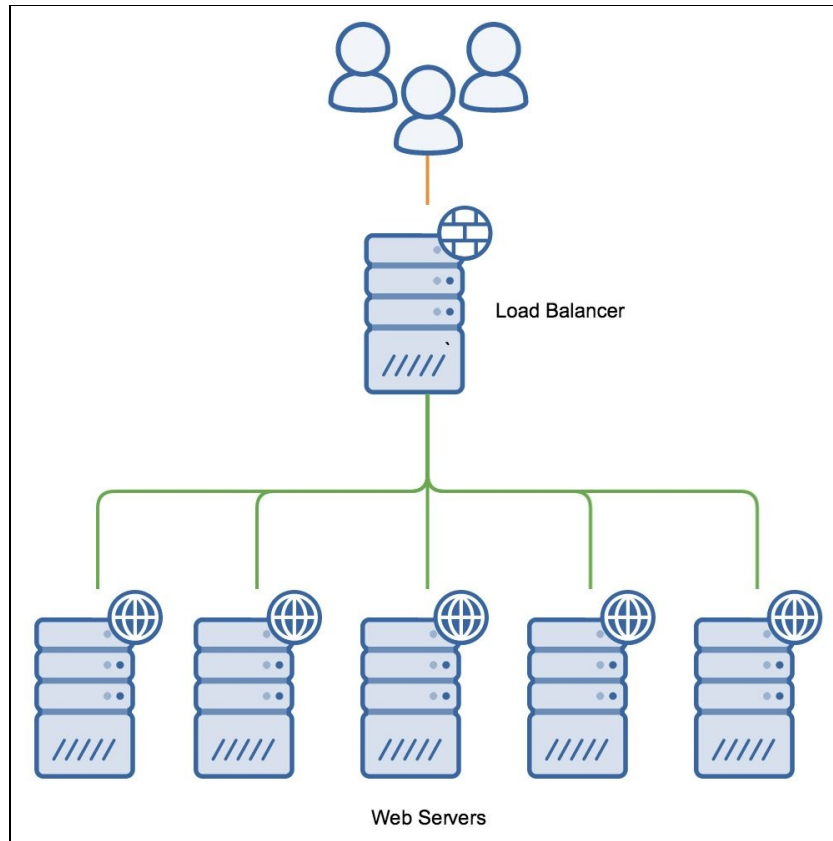
- Basic balancing techniques
- Round-robin load balancing
- Least connected load balancing
- Hash-based load balancing
- Testing and debugging NGINX load balancing
- TCP / application load balancing
- NGINX as an SMTP load balancer

Introduction

Load balancing serves two main purposes—to provide further fault tolerance and to distribute the load. This is achieved by dividing incoming requests against one or more backend servers, so that you get the combined output of these multiple servers. As most load balancer configurations are generally configured as a reverse proxy (as detailed in the previous chapter), this makes NGINX a great choice.

By increasing your fault tolerance, you can ensure the reliability and uptime of your website or application. In the realms of Google or Facebook, where seconds of downtime can cause chaos, load balancers are a critical part of their business. Likewise, if you have occasional issues with your web server, or want to be able to conduct maintenance without bringing your site down, then a load balancer will greatly enhance your setup.

The distributed load side of a load balancer allows you to horizontally scale your website or application. This is a much easier way to scale your website rather than simply throwing more hardware at a single server, especially when it comes to high levels of concurrency. Using a load balancer, we can increase the number of servers easily, as shown in the following figure:



With the right configuration and monitoring, you can also add and remove these web servers on the fly. This is what many refer to as **elastic computing**, where resources can be provisioned in an automated fashion. When implemented correctly, this can deliver cost savings, while ensuring that you can handle peak loads without any issue.

Of course, there are a few caveats here. The first is that your application or website must be able to run in a distributed manner. As your web server doesn't have a centralized filesystem by default, handling file uploads must be done in a way that all servers still retain access. You can use a clustered filesystem to achieve this (for example, GlusterFS) or use a centralized object or file storage system, such as AWS S3.

Your database also needs to be accessible by all your web servers. If your users log in to your system, you'll also need to ensure that the session tracking uses a database so that it's accessible from all servers.

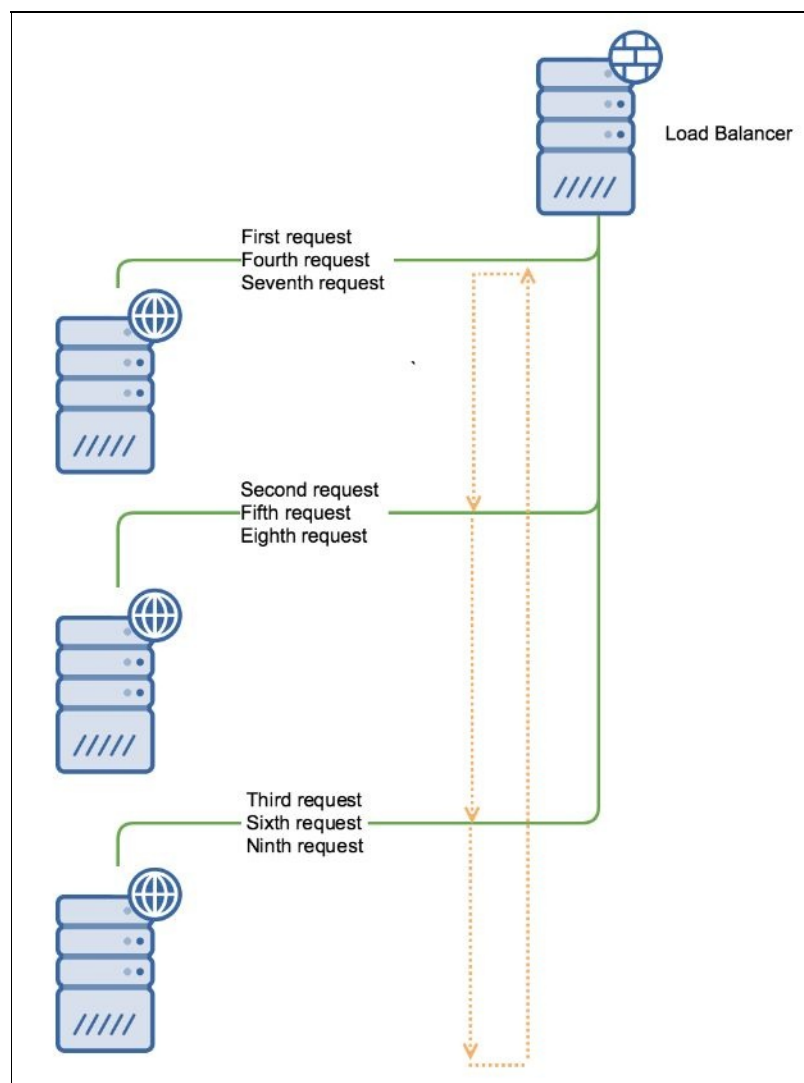
Thankfully though, if you're using a modern framework (as we covered in [Chapter](#)

3, *Common Frameworks*) or a modern **Content Management System (CMS)**, then these aspects have been previously implemented and documented.

Basic balancing techniques

The three scheduling algorithms which NGINX supports are round-robin, least connections, and hashing.

Load balancers configured in a **round-robin** fashion distribute requests across the servers in a sequential basis; the first request goes to the first server, the second request to the second server, and so on. This repeats until each server in the pool has processed a request and the next will simply be at the top again. The following diagram explains the round robin scheduling algorithm:



This is the most simplistic method to implement, and it has both positive and negative sides. The positive is that no configuration is required on the server side. The negative is that there's no ability to check the load of the servers to even out the requests.

When configured to use the **least connection** method of load balancing, NGINX distributes the requests to the servers with the least amount of active connections. This provides a very rudimentary level of load-based distribution; however, it's based on connections rather than actual server load.

This may not always be the most effective method, especially if one particular server has the least amount of connections due to a high resource load or an internal issue.

The third method supported by NGINX is the **hash** method. This uses a key to determine how to map the request with one of the upstream servers. Generally, this is set to the client's IP address, which allows you to map the requests to the same upstream server each time.

If your application doesn't use any form of centralized session tracking, then this is one way to make load balancing more compatible.

Round robin load balancing

Getting ready

To test the load balancing, you'll need to be able to run multiple versions of your app, each on different ports.

```
upstream localapp {  
  
    server 127.0.0.1:8080; server 127.0.0.1:8081; server  
    127.0.0.1:8082; }  
  
server {  
  
    listen 80; server_name load.nginxcookbook.com; access_log  
    /var/log/nginx/load-access.log combined; location / {  
  
        proxy_pass http://localapp; }  
  
}
```

How it works...

In our `upstream` block directive, we define the servers and the name of the backend servers. In our recipe, we simply defined these as three instances on the localhost on ports `8080`, `8081`, and `8082`. In many scenarios, this can also be external servers (in order to horizontally balance resources).

In our `server` block directive, instead of connecting directly to a local application, as in the previous recipe, we connect to our upstream directive which we named `localapp`.

As we didn't specify an algorithm to use for load balancing, NGINX defaults to the round-robin configuration. Each of our entries is loaded in sequential order as new requests come in, unless one of the servers fails to respond.

```
upstream localapp {
```

```
    server 127.0.0.1:8080 weight=2; server 127.0.0.1:8081; server  
    127.0.0.1:8082; }
```

Because we set the first server with a weighted value of 2, it will receive twice as many requests as the others.

See also

The NGINX upstream module: http://nginx.org/en/docs/http/ngx_http_upstream_module.html

Least connected load balancing

While the default load balancing algorithm is round-robin, it doesn't take into consideration either the server load or the response times. With the **least connected** method, we distribute connections to the upstream server with the least number of active connections.

Getting ready

To test the load balancing, you'll need to be able to run multiple versions of your app, each on different ports.

```
upstream localapp {  
    least_conn;  
  
    server 127.0.0.1:8080; server 127.0.0.1:8081; server  
    127.0.0.1:8082; }  
  
server {  
  
    listen 80; server_name load.nginxcookbook.com; access_log  
    /var/log/nginx/load-access.log combined; location / {  
  
        proxy_pass http://localapp; }  
  
}
```

How it works...

Like the round-robin configuration, we have three upstream servers defined with the name `localapp`. For this configuration, we explicitly tell NGINX to use the `least_conn` method of load balancing.

As each new request comes in, NGINX determines which upstream server has the least amount of connections and directs requests to this server.

Hash-based load balancing

When you need to ensure that hash-based load balancing is the optimal choice, commonly, the client's IP address is used as the pattern to match so that any issues with cookies and per upstream server session tracking is sticky. This means that every subsequent request from the same hash will always route to the same upstream server (unless there's a fault with the upstream server).

```
upstream localapp {
```

```
    hash $remote_addr consistent; server 127.0.0.1:8080; server  
    127.0.0.1:8081; server 127.0.0.1:8082; }
```

```
server {
```

```
    listen 80; server_name load.nginxcookbook.com; access_log  
    /var/log/nginx/load-access.log combined; location / {
```

```
        proxy_pass http://localapp; }
```

```
}
```

How it works...

For this hash method, we used the client IP (`$remote_addr`) as the determining factor to build up the hash map.

The `consistent` parameter at the end of the hash line implements the Ketama consistent hashing method, which helps to minimize the amount of remapping (and therefore potential disruption or cache loss) if you need to add or remove servers from your `upstream` block directive. If your upstream servers remain constant, then you can omit this parameter.

```
upstream localapp {
```

```
    ip_hash; server 127.0.0.1:8080; server 127.0.0.1:8081; server  
127.0.0.1:8082; }
```

While this method still works, if you need better consistency for ip_hash mapping, then using hash \$remote_addr will match the full IP address.

See also

- The consistent hashing Wikipedia page: https://en.wikipedia.org/wiki/Consistent_hashing
- The `hash` directive documentation: http://nginx.org/en/docs/http/nginx_http_upstream_module.html#hash

Testing and debugging NGINX load balancing

As load balancing can introduce extra complexities into your environment, it's important to ensure that you can test your setup thoroughly. Especially when you're trying to debug corner cases, being able to build a test platform becomes critical to reproducing faults or issues.

Ideally, you want this combined with real-time metrics as well, especially if you want to be able to overlay this with data from your upstream servers. As we'll cover in [Chapter 13](#), *NGINX Plus – The Commercial Offering*, the commercial release of NGINX contains a live monitoring module which provides information such as the current connections, upstream server statuses, and load information.

While there are a lot of programs and cloud services around which you can generate load to test your load balancer from a client perspective, I didn't find many tools to have test instances for the upstream side of the problem. So, like any programmer, I simply wrote my own!

HTest is an open source tool, which emulates a web server under varying conditions. It is written in Go and is therefore able to take advantage of the high levels of concurrency and optimization; HTest can serve more than 150,000 requests a second on very modest hardware.

Rather than just serving a static page, HTest allows you to vary the response times (so that you can emulate your typical application responses), failure rates (where a percentage of responses return a 500 error), and can also introduce some jitter so that the results are more realistic.

Getting ready

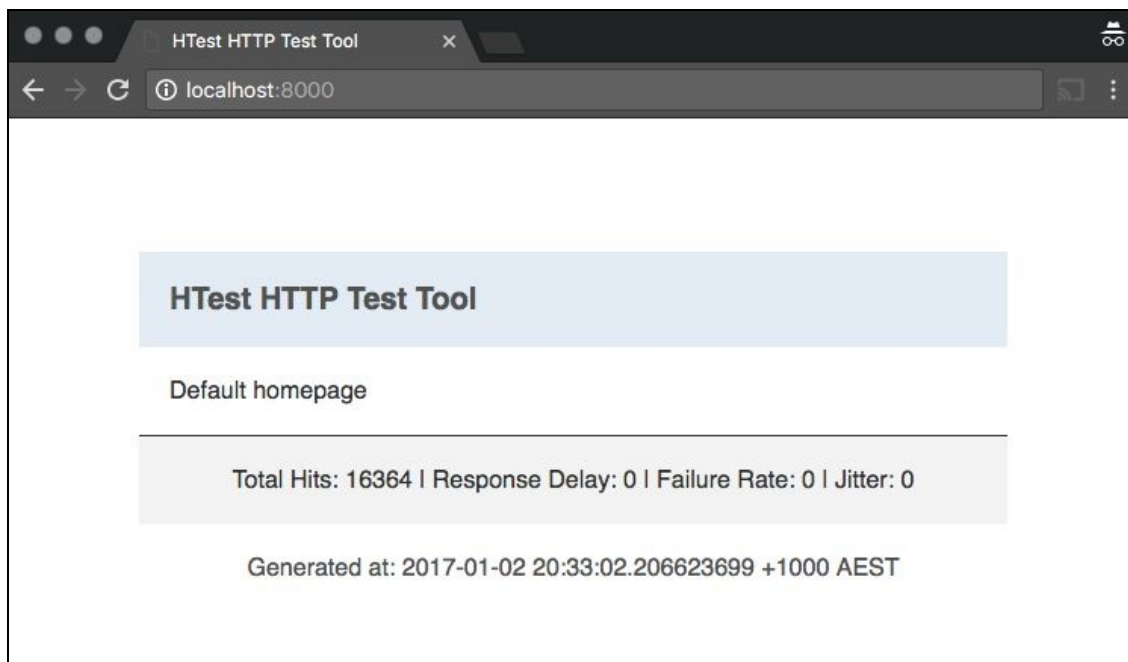
No prerequisite steps are required for this recipe.

How to do it...

After you have downloaded the latest HTest executable (within the releases section of the GitHub page), you can simply call HTest:

```
| ./htest
```

By default, HTest will listen on port 8000 on 127.0.0.1 and be ready to serve requests. There's a basic web page, which is returned as part of the result and looks like this:



As each request comes in, the hit counter will be incremented, and if any flags for delays, failures, or jitter have been set, then these will also be shown.

We can easily start HTest on different ports, which when testing a load balancer on a single server, is required:

```
| ./htest -port 8001
```

Details such as the response delay can be set in either of two ways. If you want to start HTest with a delay, this can also be done via a command-line flag:

```
| ./htest -responsedelay 20
```

This value is in **milliseconds (ms)**. Alternatively, we can also set the delay after the program has started with a simple cURL call:

```
| curl http://localhost:8000/svar/?responsedelay=10
```

The advantage to this is that it can be scripted so that you have a stepped test routine, or you can vary it manually to measure the effect it has on your overall load balancer configuration. Here's the full reference table for configuration items:

Field	Value	Example
responsedelay	Delay time (in ms)	10
failurerate	Failure rate of requests (in percent)	5
jitter	Variance of the results (in percent)	2

There's more...

To further aid in tracing and testing load balancer configurations, we can add some additional headers in for testing. This allows us to then see further information as to which upstream server processed the request, as well as how long it took. Here are the additional headers I added for testing: `add_header upstream-addr $upstream_addr; add_header upstream-connect-time $upstream_connect_time;`

If it's a production system, you can enable this only when a conditional flag is set, as we detailed in [Chapter 5, Logging](#).

This allows us to see which upstream server processed our request (`$upstream_addr`) and how long it took to connect to the upstream server (`$upstream_connect_time`). This helps to give an indication as to where any possible connection delays are occurring, and also from which upstream server.

If you need to track the upstream server response time (how long it took to return data), this needs to be logged and cannot be set as a header. This is because headers are sent to the browser before the request from the upstream server has returned, and therefore the time at that point is still unknown.

For example, here are the results from a very simple test:



As this is based on our previous recipes, we can see that the connection was made to a server running locally on port 8081, and because the load on the server was very low, there was no connection delay (0.000 seconds). An increase in

connection times can indicate load or network issues at the load balancer end, which is typically hard to diagnose, as the finger is usually pointed at the upstream server as the cause of delays.

See also

The HTest repository: <https://github.com/timbutler/htest>

TCP / application load balancing

While most people know NGINX for its outstanding role as a web and proxy server, most won't have used it beyond the standard web roles. Some of the key functionalities come from the fact that NGINX is incredibly flexible in how it operates. With the introduction of the stream module in 1.9, NGINX can also load balance TCP and UDP applications as well.

This opens up the possibility to load balance applications which don't have either any internal task distribution or any ability to scale beyond one server.

How to do it...

To use TCP load balancing, we first need to double-check whether the NGINX version has the stream module compiled. To do this, you can run the following command: **nginx -V**

This will generate an output, displaying all of the compiled modules. The stream module is available if you see `--with-stream` in the output. Consider this example:

```
root@nginx-ubuntu-14:/etc/nginx# nginx -V
nginx version: nginx/1.11.8
built by gcc 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3)
built with OpenSSL 1.0.1f 6 Jan 2014
TLS SNI support enabled
configure arguments: --prefix=/etc/nginx --sbin-path=/usr/sbin/nginx --modules-path=/usr/lib/nginx/modules --conf-path=/etc/nginx/nginx.conf --error-log-path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --pid-path=/var/run/nginx.pid --lock-path=/var/run/nginx.lock --http-client-body-temp-path=/var/cache/nginx/client_temp --http-proxy-temp-path=/var/cache/nginx/proxy_temp --http-fastcgi-temp-path=/var/cache/nginx/fastcgi_temp --http-uwsgi-temp-path=/var/cache/nginx/uwsgi_temp --http-scgi-temp-path=/var/cache/nginx/scgi_temp --user=nginx --group=nginx --with-compat --with-file-aio --with-threads --with-http_addition_module --with-http_auth_request_module --with-http_dav_module --with-http_flv_module --with-http_gunzip_module --with-http_gzip_static_module --with-http_mp4_module --with-http_random_index_module --with-http_realip_module --with-http_secure_link_module --with-http_slice_module --with-http_ssl_module --with-http_stub_status_module --with-http_sub_module --with-http_v2_module --with-mail --with-mail_ssl_module --with-stream --with-stream_realip_module --with-stream_ssl_module --with-stream_ssl_preread_module --with-cc-opt='-g -O2 -fstack-protector --param=ssp-buffer-size=4 -Wformat -Werror=format-security -Wp,-D_FORTIFY_SOURCE=2' --with-ld-opt='-Wl,-Bsymbolic-functions -Wl,-z,relro -Wl,-z,now -Wl,--as-needed'
```

If you don't have the required version, we covered how to install an updated version in [Chapter 1, Let's Get Started](#).

Next, we need to define a `stream` block directive, which must be situated outside of the HTTP block directive or replace it altogether. Unlike previous examples, where this could be within the `/etc/nginx/conf.d/` directive, this `stream` block needs to be set within the main NGINX configuration file (typically `/etc/nginx/nginx.conf`) or at least included from there and outside the `http` block directive. Here's our configuration: `stream { upstream tcpapppool { hash $remote_addr consistent; server 127.0.0.1:8101; server 127.0.0.1:8102; server 127.0.0.1:8103; } server { listen 7400; proxy_pass tcpapppool; } }`

How it works...

In our recipe, we define an `upstream` block, which is virtually identical to our HTTP load balancer configurations. We also specified a hash against the client's IP (`$remote_addr`), as any application which only has internal session tracking for authentication or similar would require re-authentication against each upstream server if new connections are made.

We have three upstream servers specified for this recipe, which again are against the loopback interface on the local server. Each instance of your TCP applications would need to be listening on `127.0.0.1` on ports `8101`, `8102`, and `8103` individually.

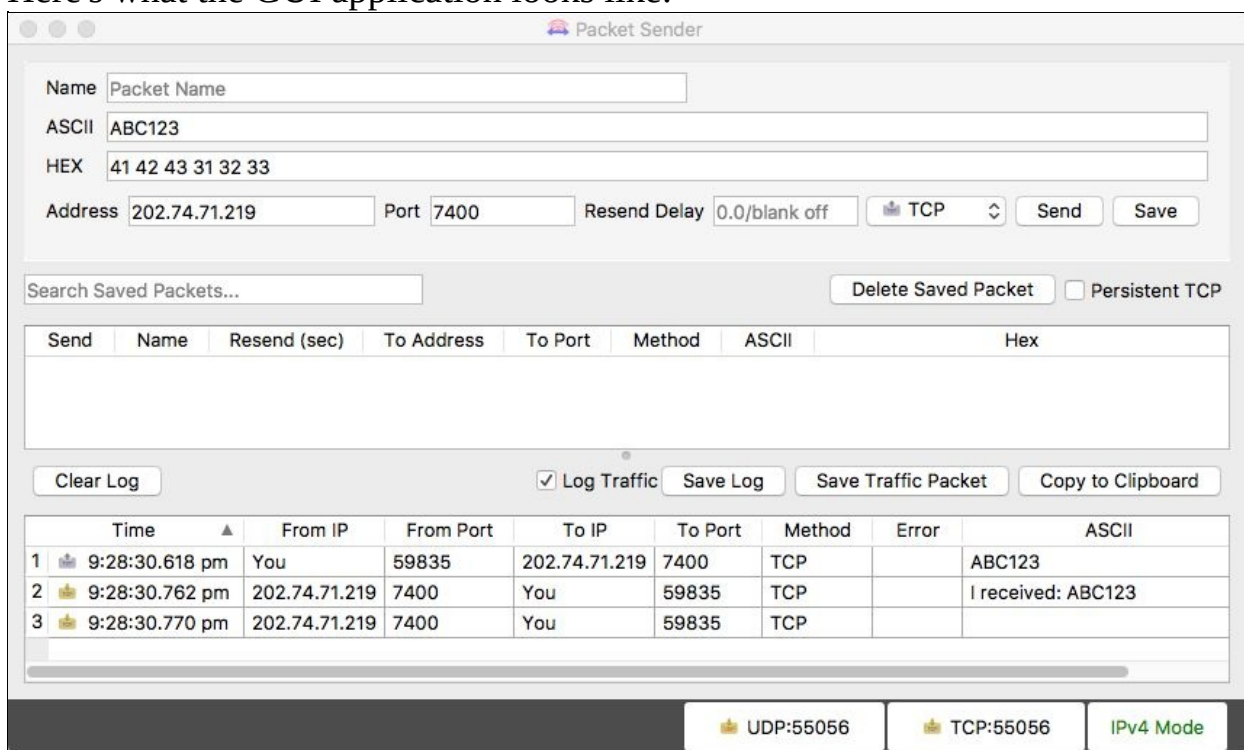
Our `server` block directive then tells NGINX to listen on port `7400`. As we haven't specified a protocol, it will default to TCP. If you require UDP, you'll need to specify the protocol as a parameter. Consider this example: `listen 7400 udp;`

Lastly, we then configure a reverse proxy (`proxy_pass`) to our named upstream configuration `tcpapppool1`.

Easy testing

If you need an application to help test connections, I'd recommend you try Packet Sender. This application is free, cross-platform, and allows you to send and receive UDP and TCP data both through a GUI and command-line interface.

This makes it perfect when testing new configurations, especially if you need either targets for your load balancer, or to test the connections through NGINX. Here's what the GUI application looks like:



As shown in the screenshot, we sent a quick ABC123 packet to the NGINX server and received I received: ABC123 back from our upstream application (which is a simple echo app).

There's more...

Just like traditional HTTP based configurations, we can also enable access logs for TCP load balancing. We can tailor the logs to suit specific applications, which may include fields such as the client IP address, bytes sent, and bytes received.



This requires NGINX version 1.11.4 or higher.

To use the stream logging, we firstly need to define a log format. Here's a basic configuration:

```
log_format basic '$remote_addr [$time_local] '
                 '$bytes_sent $bytes_received $session_time';
```

Then, we add the `access_log` directive to our `server` block:

```
server {
    listen      7400;
    access_log  /var/log/nginx/tcp-access.log  basic;
    proxy_pass  tcpappool;
}
```

This basic log format then gives us an output like this:

```
1.2.3.4 [28/Dec/2016:22:17:08 +1000] 18 6 0.001
1.2.3.4 [28/Dec/216:22:17:08 +1000] 18 6 0.001
1.2.3.4 [28/Dec/216:22:17:09 +1000] 18 6 0.000
1.2.3.4 [28/Dec/216:22:17:09 +1000] 18 6 0.001
```

We can see the originating IP (1.2.3.4), the time of the connection (28 December), the bytes sent and received, and then the total session time. If more detailed information is required (such as logging the upstream server used), the log format can be tailored to your specific needs.

See also

- The NGINX stream module documentation: https://nginx.org/en/docs/stream/nginx_stream_core_module.html
- The NGINX stream log module documentation: https://nginx.org/en/docs/stream/nginx_stream_log_module.html
- Packet Sender: <https://packetsender.com/>

NGINX as an SMTP load balancer

As demonstrated in our previous recipe, NGINX can do pure TCP load balancing. Further to this, there are some protocol specific implementations, which have some specific items to enhance the implementation.

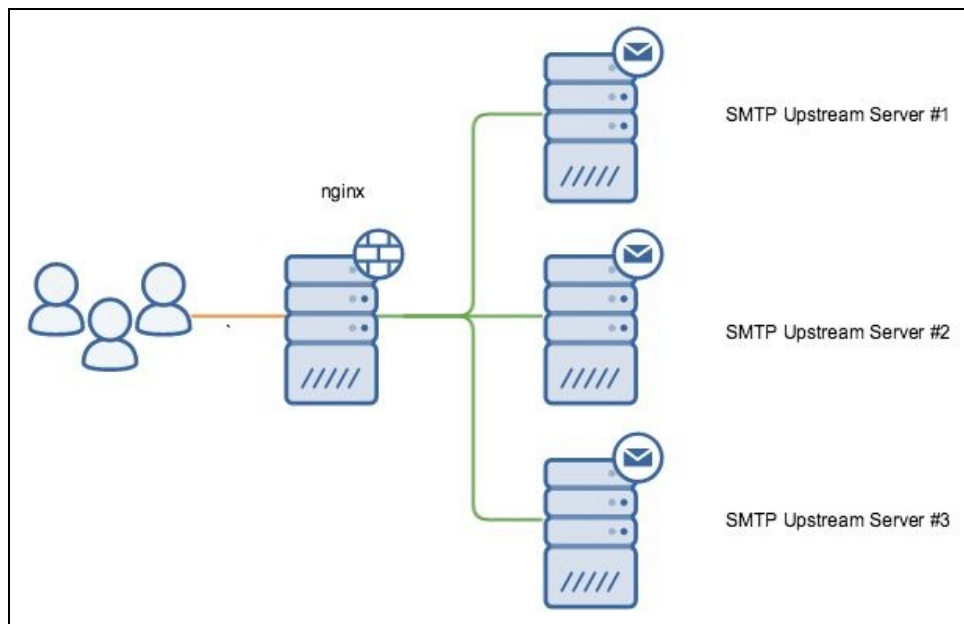
Simple Mail Transport Protocol (SMTP) is the standard protocol used to send and receive email at a server level. The most popular SMTP servers for a Linux platform include Postfix, Exim, and Sendmail, with Exchange being the most popular for Windows.

Load balancing SMTP can help to distribute the sending and receiving of email, especially if it's a high-volume environment such as an **Internet Service Provider (ISP)**. By running multiple servers, you can distribute the sending aspect as well as provide some fault tolerance when systems have issues.

While NGINX has a specific mail module, unfortunately, this does not have load balancing capabilities. The good news is, the stream module is flexible enough that it works seamlessly with SMTP.

How to do it...

We're going to configure three local SMTP applications, which will be used to help distribute the load. This is what our configuration looks like:



As this needs the stream module, we need to confirm that we have the right NGINX version first. To do this, we run this command:

```
| nginx -v
```

If you see `--with-stream` in the output, you have the required module. Otherwise, first start with the instructions in [Chapter 1, Let's Get Started](#), to install an updated version.

Then, we define our `stream` block directive, which must be at the root level and not within the `http` block, like most of the other recipes in this book. You'll need to add it to the main NGINX configuration file (typically `/etc/nginx/nginx.conf`) or at least include the configuration from here.

Here's the `stream` block directive:

```
| stream {  
|     upstream smtpool {
```

```
server 127.0.0.1:2501;
server 127.0.0.1:2502;
server 127.0.0.1:2503;
}

log_format smtplog '$remote_addr $remote_port -> $server_port '
                   '[$time_local] $bytes_sent $bytes_received '
                   '$session_time ==> $upstream_addr';

server {
    listen 25;
    proxy_pass smtppool;
    access_log /var/log/nginx/smtp-access.log smtplog;
}
}
```

How it works...

Firstly, we define our `upstream` block directive and call it `smtppool`. There are three servers within this directive, which run on the same server as the NGINX and therefore listen on `127.0.0.1`. A real-world scenario will have these running on external servers to help distribute the load. As there's no explicit load balancing method set, this will default to round robin.

Next, we defined a custom log format, which we named `smtplog`. Compared to the previous recipe's more basic format, this time we added logging for the port numbers, as well as the upstream server used.

Here's an example of what the logs produce:

```
1.2.3.4 64811 -> 26 [04/Jan/2017:23:43:00 +1000] 282 122 0.041 ==> 127.0.0.1:2501
1.2.3.4 64812 -> 26 [04/Jan/2017:23:43:00 +1000] 282 122 0.039 ==> 127.0.0.1:2502
1.2.3.4 64813 -> 26 [04/Jan/2017:23:43:00 +1000] 282 122 0.040 ==> 127.0.0.1:2503
1.2.3.4 64814 -> 26 [04/Jan/2017:23:43:01 +1000] 282 122 0.037 ==> 127.0.0.1:2501
1.2.3.4 64815 -> 26 [04/Jan/2017:23:43:01 +1000] 282 122 0.038 ==> 127.0.0.1:2502
1.2.3.4 64816 -> 26 [04/Jan/2017:23:43:01 +1000] 282 122 0.040 ==> 127.0.0.1:2503
1.2.3.4 64817 -> 26 [04/Jan/2017:23:43:01 +1000] 282 122 0.110 ==> 127.0.0.1:2501
```

While the upstream SMTP servers themselves should also have detailed logs, these logs can help when there are issues occurring and help diagnose if it's a particular upstream server at fault. We can also see that the upstream server used is different every time and in sequential order. This shows that the round robin load balancing is working as expected.

Lastly, we have our `server` block directive. We tell NGINX to listen on port 25 (the default for SMTP) and proxy connections to our `smtppool` upstream servers. We also then log the access using the log format (named `smtplog`) defined earlier.

```
server {
```

```
    listen 25; listen 485; listen 581; proxy_pass smtppool; access_log  
/var/log/nginx/smtp-access.log smtplog; }
```

With the server port (named `$server_port`) logged in our custom log format, we're still able to trace issues down to a specific port.

See also

- The NGINX stream module documentation: https://nginx.org/en/docs/stream/nginx_stream_core_module.html
- The NGINX stream log module documentation: https://nginx.org/en/docs/stream/nginx_stream_log_module.html

Advanced Features

In this chapter, we will cover the following recipes:

- Authentication with NGINX
- WebDAV with NGINX
- Bandwidth management with NGINX
- Connection limiting with NGINX
- Header modification with NGINX

Introduction

If you've read this cookbook up until this chapter, you'll know that NGINX is a very flexible and powerful platform. Even with what we covered so far, you'll know that there are a number of additional modules and extra flexibility we can use to enhance your current configuration.

Some of these recipes may be required especially once you start to scale, since servers are a finite resource.

Authentication with NGINX

While many CMSes and advanced web applications have their own authentication systems, we can use NGINX to provide a second layer. This can be used to provide multifactor authentication and also to limit brute force attempts.

Alternatively, if you have a very basic application or a system, such as Elasticsearch, without any authentication, NGINX is a natural fit to provide for this role.

Getting ready

This recipe assumes that you have an existing web application. This could be as simple as static pages or a full CMS such as WordPress.

We'll also need to install Apache utilities (not the full web server), which is generally packaged as `apache2-utils` on Debian/Ubuntu-based systems and `httpd-tools` on CentOS/RedHat-based distributions.

How to do it...

In order to provide basic authentication, we first need to create a password file. We can do this with the `htpasswd` utility, which is part of the Apache tools. It's important that we don't store this file in a publicly accessible directory, otherwise your usernames and passwords will be compromised.

Here's how to create the password file and add the `siteadmin` username to it:

```
| htpasswd -c /var/www/private/.htpasswd siteadmin
```

The `htpasswd` utility will then prompt you for a password. While we could specify this via the command line, it would also mean that your password is logged in plain text within your bash history. Ensuring that the password is entered via `stdin` reduces the risks of compromise if your log files are exposed.

Next, we can add additional users to the password file (if they're required). If I want to add one login per user, I can specify it as follows:

```
| htpasswd /var/www/private/.htpasswd userA
```

This will again prompt for the password (and the confirmation password). You can repeat this for as many users as required. Once we have the password file, we can now set the authentication within our NGINX `server` block directive:

```
server {
    listen      80;
    server_name secure.nginxcookbook.com;
    access_log  /var/log/nginx/secure-access.log combined;

    location = /favicon.ico { access_log off; log_not_found off; }

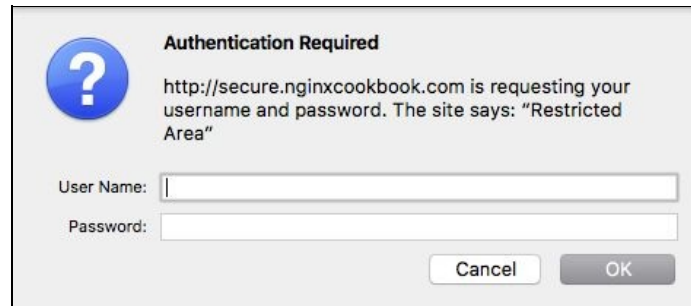
    location / {
        auth_basic "Restricted Area";
        auth_basic_user_file /var/www/private/.htpasswd;

        root    /var/www/html;
        index   index.html index.htm;
    }
}
```

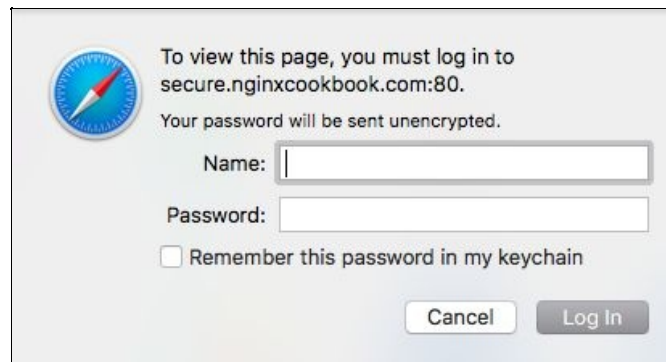


If you're not using this on an SSL encrypted site, the credentials will be transmitted in plain text. Ensure that you protect your site if you want the username and password to remain secure.

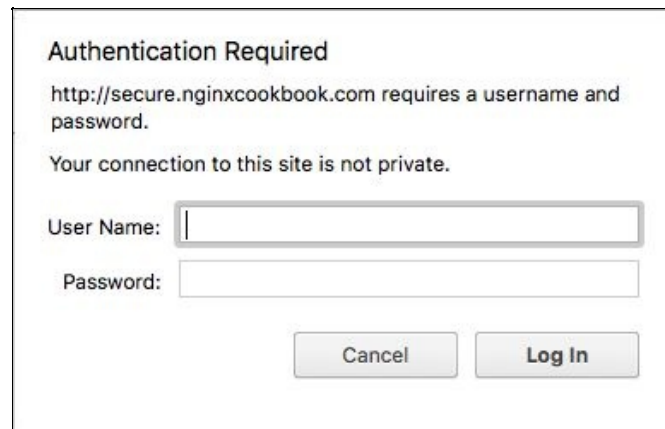
When you visit the site (in this instance, <http://secure.nginxcookbook.com>), you'll see one of the following popups. For Firefox, it should look similar to the following screenshot:



Safari users will also see a similar dialog box, requesting login details:



And finally, Chrome will also prompt you with a dialog like the following screenshot:



If you enter one of the usernames and passwords set in the previous step, you'll then be able to browse the website.

How it works...

If we take a look at the `.htpasswd` file we created, it should look something like

this: 

This file contains the username, the algorithm used (`$apr1` denotes a specific Apache MD5 implementation), the salt, and then the password. While some may be worried at the use of MD5, the `htpasswd` application iterates the password for 1,000 times to limit brute force attempts. This, combined with salting (the addition of random data) ensures that the password is very hard to brute force.

We will then define two extra directives within our `location` block directive to enable the basic authentication. The `auth_basic` directive enables authentication and the "Restricted Area" string is used as a message by some browsers.

We will then ask the authentication module to use the file we created (`/var/www/private/.htpasswd`) with the `auth_basic_user_file` directive. Again, as a reminder, make sure that this isn't in a location that can be publicly accessible from your website. Because we have set the `location root` directive to `/var/www/html`, and the password file within `/var/www/private`, it cannot be directly accessed.

```
location ~ ^(/wp-login.php/wp-admin/) {  
  
    auth_basic "Restricted Area"; auth_basic_user_file  
/var/www/private/.htpasswd; location ~ \.php$ {  
  
    fastcgi_pass unix:/var/run/php7.0-fpm.sock; fastcgi_index  
index.php; fastcgi_param SCRIPT_FILENAME <br/>  
$document_root$fastcgi_script_name; include fastcgi_params; }  
  
}
```

The configuration matches any request starting with (as denoted by the ^ symbol) either /wp-login.php or (|) /wp-admin/ and adds basic authentication. All other pages on the site don't contain any extra authentication and therefore load normally.

We also add a nested PHP location block, as NGINX won't process the declaration outside of the current location block. For neatness, you could also define this in a separate file once and then include it at each required location. This way, if you ever need to make changes, then it's only in one location.

See also

- The NGINX basic authentication module can be found at http://nginx.org/en/docs/http/ngx_http_auth_basic_module.html
- For the `htpasswd` program, refer to <https://httpd.apache.org/docs/2.4/programs/htpasswd.html>

WebDAV with NGINX

Web Distributed Authoring and Versioning (WebDAV) is an extension to the standard HTTP protocol that allows remote authoring commands, such as the ability to lock, upload, delete, and create content.

This content can be in the form of documents, images, objects, and more. While its popularity has declined with the rapid update of full CMSes and cloud storage platforms such as Dropbox, WebDAV still remains a very simple option to implement solutions.

Getting ready

The standard NGINX installation only includes basic WebDAV support; however, we can extend it to provide the full functionality by compiling a custom module. At the time of writing, the extended module hadn't been converted into a dynamic module, so we need to recompile all of NGINX.

Firstly, we will download the source for NGINX and prepare it for compilation:

```
| mkdir /tmp/nginxbuild  
| cd /tmp/nginxbuild  
| yumdownloader --source nginx
```

Then, we will download a copy of the extension so that it can be included in the source build:

```
| git clone https://github.com/arut/nginx-dav-ext-module.git /tmp/nginxbuild/nginx-dav-e
```

To include the extra module, we modify the `nginx.spec` file to compile the additional module to add the additional build requirements, copy the source code, and then modify the configure line. To add the extra library for the build process, we append the following line after `zlib-devel` and `pcre-devel`:

```
| BuildRequires: expat-devel
```

For the source to be included in the source RPM, we then specify the extra directory:

```
| Source14: nginx-dav-ext-module
```

Finally, we ensure that it's compiled in by appending the following to the `BASE_CONFIGURE_ARGS` definition:

```
| --add-module=%{SOURCE14}
```



A patch file is available in the official code repository for the book.

With the build configuration updated, we can now recompile NGINX with our

extra module:

```
| /usr/bin/mock --buildsrpm --spec /tmp/nginxbuild/nginx.spec --sources /tmp/nginxbuild  
| /usr/bin/mock --no-clean --rebuild /var/lib/mock/epel-7-x86_64/result/nginx-1.11.9-1.el7
```

This will generate the source RPM, and then the compiled binary RPM, ready for installation. We can now install the RPM using `yum`:

```
| yum install /var/lib/mock/epel-7-x86_64/root/builddir/build/RPMS/nginx-1.11.9-1.el7.centos.x86_64.rpm
```



If you need to perform the same for a Debian or Ubuntu distribution, refer to the Compiling from scratch section covered in the Quick installation guide recipe in Chapter 1, Let's Get Started.

To confirm that the updated packages are available, we can run `nginx -v` (note the capital `v`) to show the modules that NGINX was compiled with:

```
nginx version: nginx/1.11.9  
built by gcc 4.8.5 20150623 (Red Hat 4.8.5-11) (GCC)  
built with OpenSSL 1.0.1e-fips 11 Feb 2013  
TLS SNI support enabled  
configure arguments: --prefix=/etc/nginx --sbin-path=/usr/sbin/nginx --modules-path=/usr/lib64/nginx/modules --conf-path=/etc/nginx/nginx.conf --error-log-path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --pid-path=/var/run/nginx.pid --lock-path=/var/run/nginx.lock --http-client-body-temp-path=/var/cache/nginx/client_temp --http-proxy-temp-path=/var/cache/nginx/proxy_temp --http-fastcgi-temp-path=/var/cache/nginx/fastcgi_temp --http-uwsgi-temp-path=/var/cache/nginx/uwsgi_temp --http-scgi-temp-path=/var/cache/nginx/scgi_temp --user=nginx --group=nginx --with-compat --with-file-aio --with-threads --with-http_addition_module --with-http_auth_request_module --with-http_dav_module --with-http_flv_module --with-http_gunzip_module --with-http_gzip_static_module --with-http_mp4_module --with-http_random_index_module --with-http_realip_module --with-http_secure_link_module --with-http_slice_module --with-http_ssl_module --with-http_stub_status_module --with-http_sub_module --with-http_v2_module --with-mail --with-mail_ssl_module --with-stream --with-stream_realip_module --with-stream_ssl_module --with-stream_ssl_preread_module --add-module=/builddir/build/SOURCES/nginx-dav-ext-module --with-cc-opt='-O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector-strong --param=ssp-buffer-size=4 -grecord-gcc-switches -m64 -mtune=generic -fPIC' --with-ld-opt='-Wl,-z,relro -Wl,-z,now -pie'
```

If you see `nginx-dav-ext-module` in the list, the extra module is available and we're ready to proceed.

```
server {  
  
    listen 443 ssl;  
  
    server_name webdav.nginxcookbook.com;  
  
    access_log /var/log/nginx/webdav-access.log combined;  
  
    ssl_certificate /etc/ssl/public.pem; ssl_certificate_key  
/etc/ssl/private.key; ssl_protocols TLSv1 TLSv1.1 TLSv1.2;  
ssl_ciphers HIGH:!aNULL:!MD5;  
  
    location / {  
  
        auth_basic "Restricted Area"; auth_basic_user_file  
/var/www/private/.htpasswd;  
  
        root /var/www/webdav; autoindex on;  
  
        client_max_body_size 4g; client_body_temp_path /tmp;  
dav_methods PUT DELETE MKCOL COPY MOVE;  
dav_ext_methods PROPFIND OPTIONS; dav_access group:rw all:r;  
create_full_put_path on;  
  
    }  
  
}
```

How it works...

To ensure that any files transmitted remain secure, we set up NGINX using `HTTPS` (which is covered in [Chapter 4, All About SSLs](#)) so that all data is encrypted in transit. We also use the basic authentication we went through in the previous recipe, so that the files are also secured by a username and password.

The `root` directive then sets where the files are stored, in this case, `/var/www/webdav`. At this stage of the configuration, it's exactly like any other static file serving. With the `autoindex` directive explicitly set to `on`, this automatically generates an index of the files so that they can be easily browsed.

To allow larger file uploads, `client_max_body_size` is set to `4g`. If you need to upload files larger than 4 GB, you need to increase this value. Then, `client_body_temp_path` defines where the temporary files will be located while they're uploading. For this recipe, we'll set this to `/tmp`, so that any file will be temporarily uploaded to this location and then moved into the root location.

Here, `dav_methods` is set to allow the `PUT`, `DELETE`, `MKCOL`, `COPY`, and `MOVE` methods, which is all of the available methods. This gives complete control to the WebDAV client to upload, create, delete, and move files as they need.

Using the extra module we compiled, `dav_ext_methods` adds two additional extensions. The `PROPFIND` extension adds the ability to return file and directory properties in XML, which is used by a number of clients to list the files. The `OPTIONS` extension returns a list of available commands to indicate what permissions are available.

Lastly, we set `create_full_put_path` to `on`, which means that we can create files within subdirectories as well. The default for WebDAV is to only allow existing subdirectories, which makes it difficult to move existing data structures.

See also

- The NGINX WebDAV module is available at http://nginx.org/en/docs/http/nginx_http_dav_module.html
- You can visit WebDAV's official site at <http://www.webdav.org/>

Bandwidth management with NGINX

If you're serving large binary files (such as video files), it's important to ensure that you fairly distribute your available bandwidth among your users. At the same time, you must ensure that this distribution doesn't impact performance nor inconvenience users by setting restrictions that are too high.

Getting ready

The modules required are built into the NGINX core, so no upgrades or external modules are required. In this recipe, we'll serve static files, but they can be easily incorporated into your existing site.

```
server {  
  
    listen 80; server_name limitbw.nginxcookbook.com; access_log  
/var/log/nginx/limitbw.log combined; location / {  
  
    limit_rate 5m; root /var/www/html; index index.html index.htm; }  
  
}
```

How it works...

The `limit_rate` directive sets the rate at which each connection can download. This rate is set in bytes per second, not bits per second. Most internet connections are normally represented in bits per second; so, to convert, you will need to divide by eight to set. Our recipe has a limit of 5 **Megabytes per second (MBps)**, so when it comes to bandwidth in bits, we'll see 40 **Megabits per Second (Mbps)**.

We can test this limit using `wget`: **wget**
<http://limitbw.nginxcookbook.com/1000meg.test>

With our limit in place, the result is as follows:

```
HTTP request sent, awaiting response... 200 OK
Length: 1000000000 (954M) [application/octet-stream]
Saving to: '1000meg.test'

1000meg.test          100%[=====>] 953.67M  4.94MB/s   in 3m 46s

2017-02-13 22:15:36 (4.21 MB/s) - '1000meg.test' saved [1000000000/1000000000]
```

While the rate will fluctuate slightly, we can see that the overall average was 4.94 MBps, which matches our specified rate of 5 MBps.

location / {

```
    limit_rate 5m; limit_rate_after 20m; root /var/www/html; index  
index.html index.htm; }
```

The additional `limit_rate_after` directive allows the download to run at full speed for the first 20 megabytes (20m), and then rate limit after this value. Many online streaming services have this implementation in place, not just to balance between the rapid start of the streaming but also to ensure that the bandwidth is fairly shared among its users.

See also

The NGINX `limit_rate` documentation can be found at http://nginx.org/en/docs/http/ngx_http_core_module.html#limit_rate

Connection limiting with NGINX

In addition to limiting bandwidth to ensure fair and equitable access among all users, NGINX is able to place limits on the number of connections. Back in [Chapter 7, Reverse Proxy](#), we covered how to rate limit connections. While they may sound the same, connection limiting is slightly different and has different use cases. Connection limiting is used where you have long running tasks, such as downloads. The previous recipe covering bandwidth limiting only applies per connection, not per IP. We can however combine the two to ensure that each IP address can't exceed the specified bandwidth limit.

Getting ready

Like the bandwidth limiting, connection limiting is built into the core of NGINX; so no further modules are required.

How to do it...

To provide connection limiting, we first need to define a shared memory space to use for tracking. This needs to be done outside of the `server` directive and generally placed in the main NGINX configuration file (`nginx.conf`). Here's our directive:

```
| limit_conn_zone $binary_remote_addr zone=conlimitzone:10m;
```

Then, we incorporate this into our `server` block directive:

```
server {  
    listen 80;  
    server_name limitcon.nginxcookbook.com;  
    access_log /var/log/nginx/limitcon.log combined;  
    location / {  
        root /var/www/html;  
        limit_conn conlimitzone 1;  
        limit_rate 5m;  
        index index.html index.htm;  
    }  
}
```

We can confirm this by downloading a large file via the browser and then opening another tab. If you navigate to any other page from the same site, you should see the following:



Until the first request is completed, all subsequent attempts will display a 503 error.

How it works...

We create a shared memory space with the `limit_conn_zone` directive so that the connections can be tracked. We use `$binary_remote_addr` to track the remote (client) IP address and then name our zone `conlimitzone`. Finally, we allocate 10 MB by appending the zone name with `:10m`. This is the total space allocation, which is enough to track up to 160,000 concurrent IP addresses.

In our `server` block directive, we then use our zone by setting `limit_conn` to the zone name `conlimitzone`. This limit is then set to a total of `1` connection, which ensures that each unique IP address is only allowed to make a single connection.

Finally, we use `limit_rate` (as shown in our previous recipe) to limit the bandwidth per connection. As we have set the connection limit to `1`, this means that each user can only download one file at a time with a total bandwidth of 5 MBps.

There's more...

Tracking connections via the client IP address isn't the only way we can limit connections. We can also set a limit for the server as a whole, which can be handy for preventing the server from being overloaded. Especially, where the system is behind a reverse proxy, this can be a simple yet effective way of ensuring that your website or application remains responsive.

To track for the server as a whole, we again first set the shared memory zone:

```
| limit_conn_zone $server_name zone=serverlimitzone:10m;
```

Then, we set our `server` directive:

```
| server {  
    listen 80;  
    server_name limitcon.nginxcookbook.com;  
    access_log /var/log/nginx/limitcon.log combined;  
    location / {  
        root /var/www/html;  
        limit_conn serverlimitzone 500;  
        index index.html index.htm;  
    }  
}
```

With this configuration, our server is limited to 500 concurrent connections. If there are more connections attempted, a 503 error will be returned to the client.

See also

The NGINX `limit_conn` module documentation can be found at http://nginx.org/en/docs/http/ngx_http_limit_conn_module.html

Header modification with NGINX

With more complexities in your system, sometimes some additional debug or information sent in the HTTP headers can be invaluable. The HTTP headers can have a specific meaning that tells the browser to treat a response a certain way or they could simply be to provide extra information, which can be used to trace specific issues from your website or applications.

Getting ready

The ability to modify the headers is already inbuilt into the NGINX core, but to use the latest features, you'll need version 1.10.0 or higher.

How to do it...

There are a number of different ways that we can modify the headers to suit different requirements. While the process remains the same for each, specific examples of where headers are commonly modified are detailed in the following examples.

Caching static content

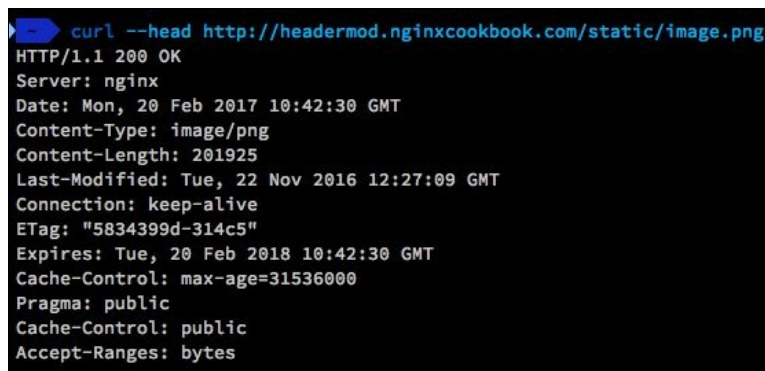
For sites where the static media files (such as CSS, JS, and images) have a version number in their filename, we can easily cache the files with a long expiry without causing any issues. This means that, unless a user clears their cache, they will have a copy of this file to speed up page reloads. To set the caching, we will use a `location` block directive (within your main `server` block directive) to add the additional headers. Here's the code required:

```
location /static {  
    expires 365d;  
    add_header Pragma public;  
    add_header Cache-Control "public";}
```

To see the headers, we can use Chrome **Developer Tools (DevTools)** or a command-line tool such as cURL. If you want to run this via cURL, here's how to do it:

```
| curl --head http://headermod.nginxcookbook.com/static/image.png
```

This will just display the headers from the server response, which will give us an output similar to the following screenshot:



```
curl --head http://headermod.nginxcookbook.com/static/image.png  
HTTP/1.1 200 OK  
Server: nginx  
Date: Mon, 20 Feb 2017 10:42:30 GMT  
Content-Type: image/png  
Content-Length: 201925  
Last-Modified: Tue, 22 Nov 2016 12:27:09 GMT  
Connection: keep-alive  
ETag: "5834399d-314c5"  
Expires: Tue, 20 Feb 2018 10:42:30 GMT  
Cache-Control: max-age=31536000  
Pragma: public  
Cache-Control: public  
Accept-Ranges: bytes
```

From the cURL output, we can see that the headers (`Expires`, `Pragma`, and `Cache-control`) have been set correctly.

Removing server name and version

By default, NGINX will set a `server` response header, which will contain the product name and version number. While it's mostly a minor thing, some see this as a leakage of information which gives hackers a potential starting point to look for attack vectors. We can remove this version number to remove the version data from the header. Here's how to do it: `server { listen 80; server_name headermod.nginxcookbook.com; server_tokens off; }`

The `server_tokens` directive is set to `on` by default, so we set it to `off` in order to disable the version number. This also removes the version number from the error pages (such as the 404 error page) as well.

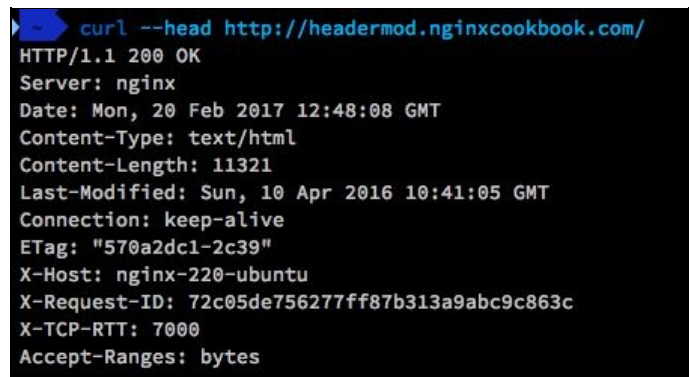
If you need to completely remove the `server` header, you'll either need a third-party module or the Plus (paid) edition of NGINX, which allows you to override it.

Extra debug headers

While you may not have looked closely, many of the top websites add additional header information to assist with debugging. Facebook adds an `x-fb-debug` header, Twitter has an `x-transaction` header, and sites such as <https://www.wired.com/> insert the `x-served-by` headers to help trace what proxies your request has passed through. This information doesn't have any impact on the end user; however, the information is invaluable when trying to diagnose hard-to-diagnose cases. While much of the low-level debug information can only be produced within your application code, at a higher level, you can easily incorporate a few helpful headers. Here's an example:

```
add_header X-Host $hostname;  
add_header X-Request-ID $request_id;  
add_header X-TCP-RTT $tcpinfo_rtt;
```

These simply need to be inserted into your existing `server` block directive, and they will generate an output like the following:



```
curl --head http://headermod.nginxcookbook.com/  
HTTP/1.1 200 OK  
Server: nginx  
Date: Mon, 20 Feb 2017 12:48:08 GMT  
Content-Type: text/html  
Content-Length: 11321  
Last-Modified: Sun, 10 Apr 2016 10:41:05 GMT  
Connection: keep-alive  
ETag: "570a2dc1-2c39"  
X-Host: nginx-220-ubuntu  
X-Request-ID: 72c05de756277ff87b313a9abc9c863c  
X-TCP-RTT: 7000  
Accept-Ranges: bytes
```

If you have a meaningful server name (which could include location, variant, cloud provider, or anything similar), adding the `$hostname` server (which we set as the `X-Host` header) allows you to trace requests down to the exact system causing the issue.

The `$request_id` command generates a 16-byte unique identifier, which can provide easy tracing back to a custom log file format (refer to [Chapter 5, Logging](#), for more information). This gives us an exact string to match, which is much more effective than trying to search log files for a date range.

Lastly, we log the TCP **Round Trip Time (RTT)** as `X-TCP-RTT`, which gives us an indication of the network performance between the server and the client. The RTT is measured in microseconds (not milliseconds) and is based on the underlying operating system's `TCP_INFO` data.



By default, these additional headers will only be inserted for 20x and 30x responses. If you want them inserted into your error pages (such as a 404), you need to append `always` to the end of the `add_header` directive.

See also

The NGINX headers module can be found at http://nginx.org/en/docs/http/nginx_http_headers_module.html

Docker Containers

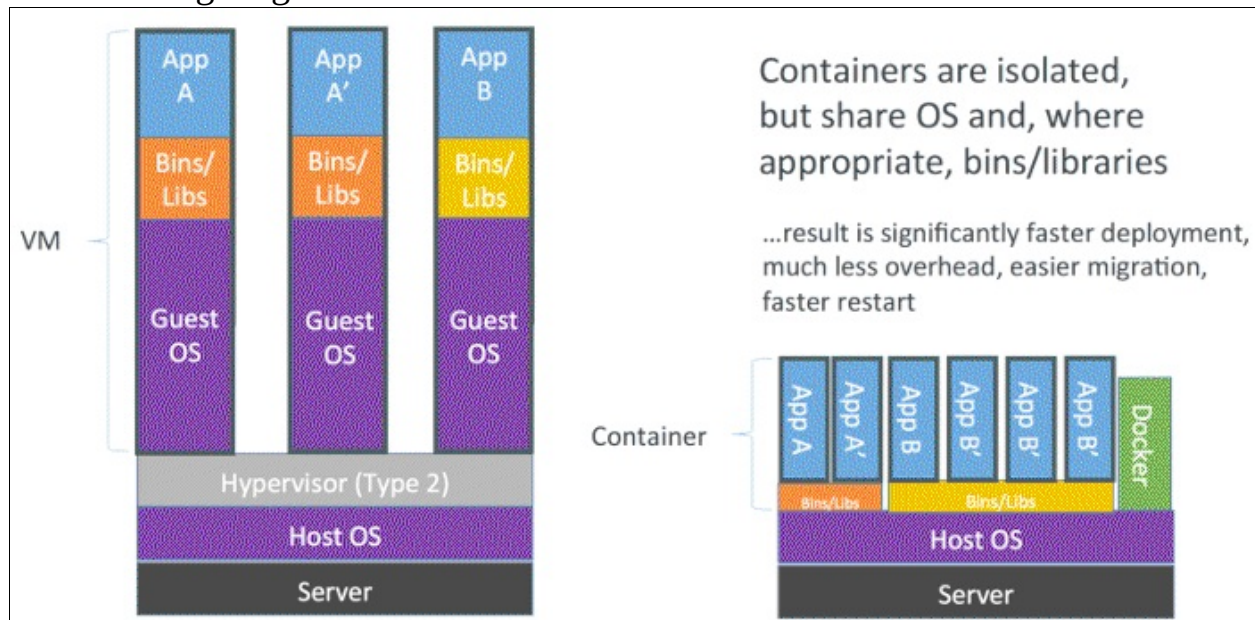
In this chapter, we will cover the following topics:

- NGINX web server via Docker
- NGINX reverse proxy via Docker
- Docker Compose with NGINX
- NGINX load balancing with Docker

Introduction

While the concept of container technology isn't new, the rise of Docker's popularity is simply because it was the first to bring simplicity and scalability to the market. For those who don't quite understand what Docker is, on a simplistic level, it's just an application container.

Containers themselves are simply another way of virtualizing your environment. Rather than having to emulate components such as a **Virtual Machine (VM)**, containers run on a single kernel and rely on software-level abstraction and isolation to provide lightweight and faster virtual environments. Docker takes this a step further and isolates it right down to a single application, as shown in the following diagram:



The advantage is that this high level of abstraction means that you can provide a highly consistent and rapidly deployable service architecture that is simple to run. Not only that, Docker has a number of tools to orchestrate and help manage the deployment of the containers as well.

Installing Docker

If you haven't installed Docker yet, the easiest way to try it is on your local development machine. Rather than running it natively, these environments set up a small Linux VM and provide wrappers to directly use it from your existing operating system.

To install it, download and run it through the installers from here:

- **Windows:** <https://www.docker.com/docker-windows>
- **macOS:** <https://www.docker.com/docker-mac>



As Docker is a rapidly evolving platform, always consult the official documentation in case there have been any changes since the time of writing.

If you're installing Docker on a dedicated server or VPS, this can be done using the standard tools. For a CentOS 7 system, this is as simple as the following:

```
| yum install docker  
| systemctl start docker
```

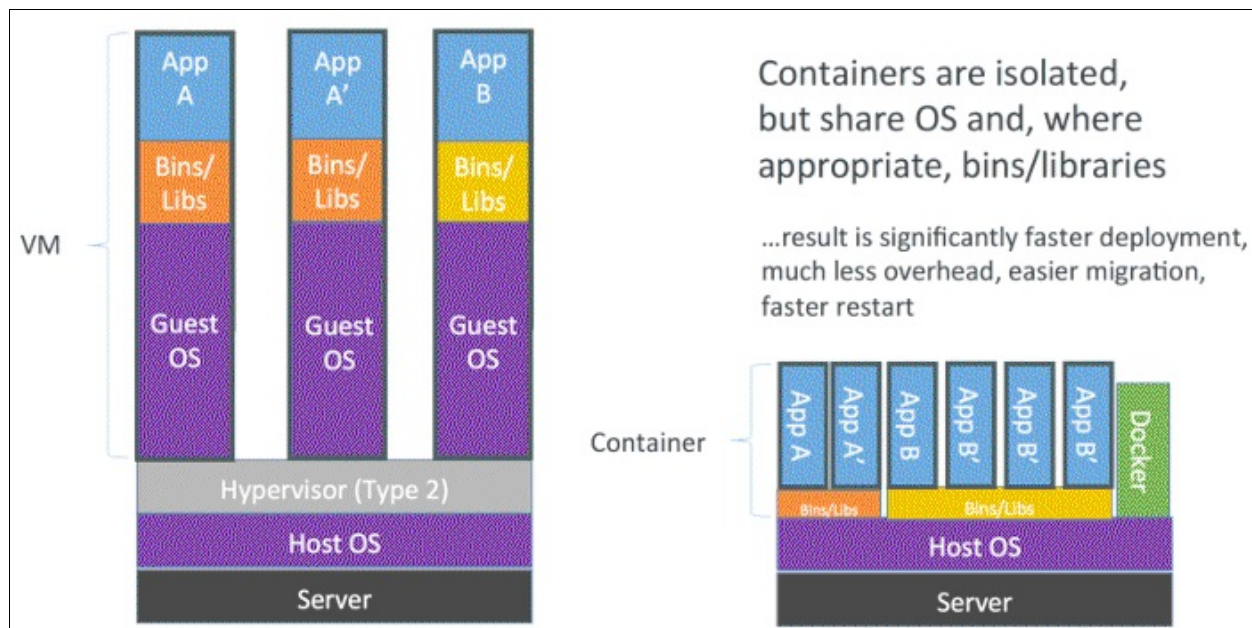
Those who are running Ubuntu, the commands are just as simple:

```
| apt install docker
```

This gives us a fully working Docker environment, even better if you have a local development installation and a staging/production system to test these recipes on.

To familiarize yourself with Docker, we'll go through a few basic commands. The first is to simply run the `docker` command. This runs the Docker client and will spit out a list of commands that are available. If you see an error at this point, double-check the output from your installation or system logs.

Next, let's run `docker ps`, which will list all the containers:



As this is a brand-new installation, you won't see any listed here. Lastly, we can run `docker version` to list both the server and client versions:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES

While the Docker client can run on a different system to the Docker server (or Docker Engine as it can also be referred to as), you'll need to ensure that the versions are compatible.

NGINX web server via Docker

This recipe will step you through the basics of setting up a simple NGINX container via Docker. When you see the simplicity of the installation, don't be fooled by how easy it is, as that's the point of Docker.

Getting ready

If you're already running existing Docker containers, make sure they're either stopped or are not running on port 80. Otherwise, they will conflict with this recipe.

How to do it...

We'll start by pulling down the latest image of Docker. While this is an optional step, it will allow you to see how Docker works in stages for the first time. To download the NGINX image (which for the officially packaged version is simply called `nginx`), run the following:

```
| docker pull nginx
```

This will then pull down a number of images, each of which will display a series of unique image IDs, like the ones displayed in this example:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Once our image has finished downloading, we can start creating our first container:

```
| docker run --name nginx-basic -d -p 81:80 nginx
```

If this works, you should get a single-line output, which will be the ID of the container:

```
docker version
Client:
 Version:      17.03.0-ce
 API version:  1.26
 Go version:   go1.7.5
 Git commit:   60ccb22
 Built:        Thu Feb 23 10:40:59 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.03.0-ce
 API version:  1.26 (minimum version 1.1)
 Go version:   go1.7.5
 Git commit:   3a232c8
 Built:        Tue Feb 28 07:52:04 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

This indicates that the container has been created and started. If this is the first

time you're seeing a containerized system, less than a second setup may seem like a fault at first. However, that's how quickly a container can be created and be up and running. To test the site, we can now browse the IP/hostname of where your Docker instance is running and connect to port 81 to confirm:

```
➤ docker version
Client:
 Version:      17.03.0-ce
 API version:  1.26
 Go version:   go1.7.5
 Git commit:   60ccb22
 Built:        Thu Feb 23 10:40:59 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.03.0-ce
 API version:  1.26 (minimum version 1.1)
 Go version:   go1.7.5
 Git commit:   3a232c8
 Built:        Tue Feb 28 07:52:04 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

How it works...

Docker images are a layered format, which allows each layer to be reused between other images. This is to save on space, rather than having 20 instances of NGINX running on Ubuntu with all the system libraries duplicated. Docker will split these instances into layers so that you can have a singular base image, and each change to this is stored as a layer. Our NGINX image looks like this:

```
➤ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
693502eb7dfb: Downloading [=====>] 19.92 MB/51.36 MB
6decb850d2bc: Download complete
c3e19f087ed6: Download complete
```

Each layer represents a change, and even the base image itself can be made up of multiple layers. Once you have a running container, any changes are then performed as **Copy-on-Write (COW)**; this means the original images are preserved, while also allowing the ability to modify instances based on them.

Our one-line deploy command can be broken down into a number of different parts:

- `run`: This tells Docker to create an instance and run the default command. As shown in the preceding figure, this is the `nginx` command to start NGINX.
- `--name nginx-basic`: This gives our container a name, which allows you to easily identify and link to it from other containers.
- `-d`: This option detaches the container from the command line and runs it in the background.
- `-p 81:80`: Using `-p` specifies port mapping for the container. It's always in this format: `<host>:<container>`. For our instance, we've opened port 81 on our server (or the development machine) and mapped it to port 80 in the container (where NGINX listens by default).
- `nginx`: Finally, we specify the image name. This can also include a version tag, or it will select the latest release if no tag is specified. If we wanted to run an older version of NGINX, we could specify `nginx:1.9.14` to use the latest 1.9 release.

There's more...

Our initial example is fairly basic, but can be easily extended to serve static files. Rather than building an image with the files deployed, we can map a volume to the host in which Docker is on. This way, we can edit the files locally but still have them served from within the Docker container. Here's our updated `run` command: **`docker run --name nginx-basic -d -p 81:80 -v /var/static:/usr/share/nginx/html:ro nginx`**

Docker's filesystems use a union filesystem (for example, OverlayFS); this allows you to join two systems at a specified mount point. In our preceding example, we mounted `/var/static` on our local server and specified the mount point as `/usr/share/nginx/html`. We've also specified that the mount is read-only to prevent anything within the container from modifying the files.

Changes to the files done within your local server (or the development machine) and in the `/var/static` directory will be served by our Docker instance.

This also means that you can keep your Docker configurations common between varying configurations and simply update the content separately. If you're especially using a **Revision Control System (RCS)** such as Git, it means you have a system that can be quickly updated via a **Continuous Integration (CI)** system.

See also

- Docker documentation: <https://docs.docker.com/>
- Official NGINX Docker hub entry: https://hub.docker.com/_/nginx/
- Official image build code: <https://github.com/nginxinc/docker-nginx>

NGINX reverse proxy via Docker

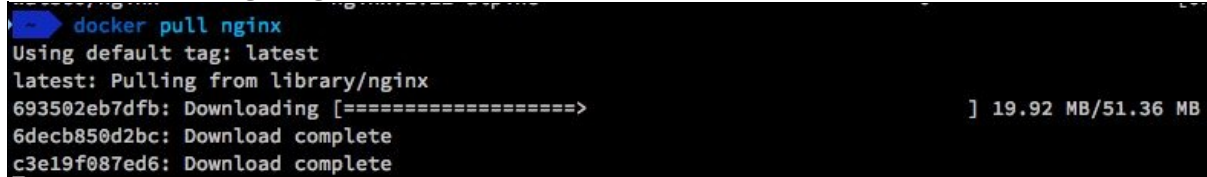
In most scenarios, Docker will be deployed alongside an application container, such as Ruby on Rails, WordPress, or similar. In traditional deployment scenarios, these would all be configured on one server. However, in a Docker-based environment, you may want to reduce each container to a single task or process where possible, like a microservice-based architecture. This is so that you can independently upgrade or replace each part without affecting the other. An example of this is updating system libraries or deploying different PHP versions. As each task is a separate container, it remains isolated and, therefore, unaffected by other containers.

Using a reverse proxy on your local development server can also be a great way to test your application before deploying. Generally, if you have a basic WAMP (that is, Windows, Apache, MySQL, PHP) style development environment, then you may not discover unique issues, which only show when you have a proxy server that mimics your production environment.

Getting ready

This recipe assumes you have some form of web or application server running on your local server. Ideally, this could be *Dockerized* as well, but we'll cover both scenarios.

Here's how it's going to look:

A terminal window with a dark background and light blue text. The command 'docker pull nginx' is entered. The output shows the default tag 'latest' being pulled from the library/nginx repository. A progress bar for the layer 693502eb7dfb shows it is downloading, with a progress indicator of ']' and '19.92 MB/51.36 MB'. The other two layers, 6dec850d2bc and c3e19f087ed6, are marked as 'Download complete'.

```
➤ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
693502eb7dfb: Downloading [=====>] 19.92 MB/51.36 MB
6dec850d2bc: Download complete
c3e19f087ed6: Download complete
```

As most real-world deployments are via HTTPS, we're also going to incorporate SSL certificate deployments.

How to do it...

Because the standard NGINX image is perfect for more complex deployments, we're going to modify it to suit our needs. The great thing about Docker is that this process is very simple to do.

Firstly, we're going to create a Docker image definition file, which is called `Dockerfile`. This is what Docker uses to build an image, and it can reference an existing image as the base so that you don't have to reinvent the wheel. Here's our `Dockerfile`:

```
FROM nginx:latest

# Configuration
COPY default.conf /etc/nginx/conf.d/

# SSL Certificates and DH Key
COPY dockerdemo.crt /etc/ssl/dockerdemo.crt
COPY dockerdemo.key /etc/ssl/dockerdemo.key
COPY dh4096.pem /etc/ssl/dh4096.pem

# Symlink the logs to stdout and stderr
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

# Expose port 443 (HTTPS)
EXPOSE 443

CMD ["nginx", "-g", "daemon off;"]
```

In the same directory, we will also need our NGINX configuration file. As we want to override the default settings, we have called this `default.conf` so that it copies over the existing file. Based on the *Configuring NGINX as a simple reverse proxy* recipe back in [Chapter 7, Reverse Proxy](#), our configuration will look like this:

```
server {
    listen          443 ssl http2;
    server_name     dockerdemo.nginxcookbook.com;
    ssl_certificate  /etc/ssl/dockerdemo.crt;
    ssl_certificate_key /etc/ssl/dockerdemo.key;
    ssl_dhparam     /etc/ssl/dh4096.pem;
    ssl_protocols   TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers     HIGH:!aNULL:!MD5;

    ssl_prefer_server_ciphers on;
    client_max_body_size 75M;
}
```

```
location / {
    proxy_pass http://127.0.0.1:8000;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header Host $host;
}
```

Lastly, we also need our SSL certificates and keys copied over as well. If you're intending to distribute this image or update the SSL certificates separately to the image, you can remove this from the image and use a volume mount to store the certificates on the local server.



If you need to generate a test SSL certificate, there's a quick guide available in Chapter 4, All About SSLs.

Once you have all the configuration files ready, you can now tell Docker to build an image. It will store this image locally, based on the tags you provide. To create the image, run the following:

```
| docker build -t nginx-demo-proxy .
```

Docker will then go through each of the steps in `Dockerfile` to produce a new image. Your output should look similar to this:

```
docker run --name nginx-basic -d -p 81:80 nginx
2ee19a3602b37e31bc2e5647778f0f01436816d11a83f2200dc2c4046af08805
```

We can see that, in the final output, our Docker image has been given the `b4007604b77e` ID. We can confirm this by viewing which Docker images we have installed:

```
| docker images
```

The following is the output listing our Docker image:

```
docker run --name nginx-basic -d -p 81:80 nginx
2ee19a3602b37e31bc2e5647778f0f01436816d11a83f2200dc2c4046af08805
```

Although there are a number of intermediate images, by default, Docker doesn't display them.

After building and confirming that we have our Docker image, you can now deploy it:

```
| docker run --name nginx-proxy -d --net=host nginx-demo-proxy
```

This will create a container based on our new image, exposing port 443 via the server. Rather than being a bridged network (therefore using docker0), we tell NGINX to use the host's network.



Currently, there's a longstanding issue with Docker for Mac accessing the host, based on the underlying OS limitation. For now, the easiest workaround is to only use container to container networking.

This is best used for development only, as you're limited to only running one instance of each image due to the lack of port mapping.

How it works...

Within our `Dockerfile`, we define a number of steps to build the image:

- **FROM:** This defines our starter image; for this recipe, we used `nginx:latest`. Rather than build the image from scratch, we simply start with the official NGINX image. If you want to manually install your own build, you could start with a base Linux, such as Debian or Alpine.
- **COPY:** In order to make the files part of the image, we can copy them as part of the build process. For this image, we've copied over our NGINX configuration file as well as the SSL certificates. If the files exist in the base image, they will simply be overwritten.
- **RUN:** We can issue commands within the build. In this instance, we symlink the default log files through to `/dev/stdout` and `/dev/stderr` so that we can view the logs from the standard Docker log tools.
- **EXPOSE:** In order to access network ports outside of the container, they must be exposed to the Docker networking subsystem. From here, they can be explicitly mapped using `-p` in the `docker run` command or implicitly mapped simply with `-P`.
- **CMD:** This defines the default command executed when the container starts. It's set up in the format of `['executable', 'param1', 'param2']`, so for our NGINX command, it translates to `nginx -g daemon off;`. Because Docker requires the application to stay in the foreground, the `-g` option allows us to set an additional directive of `daemon off` to enforce this.

There's more...

The recipe wasn't about how production systems are deployed, as we were mixing host-deployed services with container-based ones. Most production deployments of Docker have 100 percent of services deployed within Docker containers, which simplifies networking.

In a typical production deployment, here's what we might see:



With everything containerized and isolated, we can still connect these without having to expose the ports to the world. Previously, this was called **linking** containers, and while the linking commands worked, they also had significant limitations. Instead, we now create distinct networks within the host.

These networks allow containers to talk to each other in distinctly named networks and you can have multiple networks within one system without any issues. If you've previously used Docker and haven't moved to a version higher than 1.9, it's worth it for the improvements in this part alone.

To allow two containers to talk to each other, we will first create a network:

```
| docker network create webnet
```

I've named this network `webnet`, which will be between NGINX and the application only. In the preceding example, we can also create separate networks

between the Rails application and PostgreSQL, then again for Rails and Redis. This level of isolation helps ensure there's no accidental data leakage if there is a security fault.

I have installed a Redmine container (which is a project management application based on Ruby on Rails), which will benefit from a reverse proxy in front to provide SSL termination. Because of the power of Docker, we can quickly deploy a Redmine server and automatically connect it to our `webnet` network:

```
| docker run -d --name demo-redmine --net webnet redmine
```

In order for NGINX to proxy the `redmine` container, first we'll need to update the proxy configuration. When added to a network, Docker's internal DNS server will automatically add the entry so that we can simply refer to the container by name. In the `default.conf`, update, the `proxy_pass` line will look like this:

```
| proxy_pass http://redmine-demo:3000;
```

As previously, we'll need to rebuild our image and then run a container, this time linked to `webnet` instead of the host:

```
| docker build -t nginx-demo-proxy .  
| docker run --name nginx-proxy -d --net webnet -p 443:443 nginx-demo-proxy
```

When deploying the `nginx-proxy` image this time, we will also join it to the `webnet` network and then bind port 443 to the main host. Because the NGINX container is on the same network as the Redmine demo, it can explicitly access it via port 3000, whereas the host server can't. If you have everything configured correctly, you should see the proxied connection to Redmine:

49.0 MB	<div> <div> debian latest jessie 8 8.7 </div> <div> What's this? </div> </div>
	<div> <div>49.0 MB</div> <div> <div>ADD file:41ac8d85ee35954bf6c8353d9681a045ba260aa9a96...</div> <div>CMD ["/bin/bash"]</div> </div> </div>
	<div> <div>MAINTAINER NGINX Docker Maintainers "docker-maint@[hidden]"</div> <div>ENV NGINX_VERSION=1.11.10-1~jessie</div> </div>
19.3 MB	<div> <div>RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6...</div> </div>
194 bytes	<div> <div>RUN ln -sf /dev/stdout /var/log/nginx/access.log && ln -sf /dev/std...</div> <div>EXPOSE 443/tcp 80/tcp</div> </div>
32 bytes	<div> <div>CMD ["nginx" "-g" "daemon off;"]</div> </div>

See also

- macOS network limitation: <https://docs.docker.com/docker-for-mac/networking/#there-is-no-docker0-bridge-on-macos>
- Dockerfile best practices: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
- Dockerfile reference: <https://docs.docker.com/engine/reference/builder/>
- Docker networking: <https://docs.docker.com/engine/userguide/networking/>

Docker Compose with NGINX

In our previous recipes, we deployed Docker containers in a singular fashion. While this is okay for smaller projects and testing, for production environments, ideally, we want this to be as repeatable as possible. This is where Docker Compose comes into the picture. Docker Compose is a tool that allows you to define multicontainer Docker applications for ease of management and ease of deployment. It does this via a single configuration file, which defines both the containers to deploy as well as the networking.

Getting ready

Docker Compose is installed by default for all modern Docker installations. We'll take our Redmine deployment and convert it back into a one-command deployment process again.

How to do it...

To get started, we need to create a `docker-compose.yml` file. This is in YAML format, which is a simple text-based language with a strong focus on readability. Here's what our `docker-compose.yml` file looks like: `version: '3' networks: webnet: services: redmine: image: redmine networks: webnet: aliases: - demo-redmine nginx-proxy: build: ./ ports: - 443:443 networks: - webnet`

This is located in the same directory as `Dockerfile` and the previous NGINX configuration files we used. The directory naming is also important, as Docker Compose will use it to prepend the names of the containers it creates. For this recipe, I have the files located in the `composedemo` directory.

With our Docker Compose configuration file, we can now build and create the containers:

```
| docker-compose up
```

This will firstly build our `nginx-proxy` image and then proceed to create all the containers. You should see an output like this as the containers start:

```
| Creating network "composedemo_webnet" with the default driver
| Creating composedemo_redmine_1
| Creating composedemo_nginx-proxy_1
| Attaching to composedemo_nginx-proxy_1, composedemo_redmine_1
```

The naming is reflective of the configuration we used, where it prepends the directory name, adds the container name, and then appends a sequence number.

After the containers have been started, all the logs will output to the screen, and the containers will be running in the foreground. If you want to start it in the background, you can do this with the following: **`docker-compose run -d`**

This starts the containers in the background, much like our previous recipe. They can also be stopped with one command:

```
| docker-compose stop
```

With the ability to cleanly define the multicontainer deployments and then

deploy with a single command, Docker Compose is an important part of the Docker ecosystem.

How it works...

The YAML configuration file for Docker Compose has a number of elements in this recipe. Firstly, we define the version number (`version: '3'`). This isn't a revision number for the individual file, but tells Docker Compose what format to expect the configuration file in.

Next, we define the network (`webnet`). By default, Docker Compose will create a separate network for all the containers, but we have explicitly named it in this recipe. The reason we do this is to retain compatibility with our previous recipe.

Lastly, we define our services. The first is the `redmine` service, which we create from the `redmine` image. This is then added to the `webnet` network, and we also alias the name of the container. Again, this is to help maintain compatibility and reduce changes. While this isn't necessary to do, the reduction in changes—if you've come from a previous configuration—can help with diagnosing any issues.

The second service defined is our NGINX container, named `nginx-proxy`. Instead of using an image, we tell Docker Compose to build the container first from the current directory (`.`). Where we previously had to manually build and tag the image, Docker Compose does this automatically for us. We then map the host port `443` to the container port `443` and, like the Redmine service, we add it to the `webnet` network.

See also

- Docker Compose documentation: <https://docs.docker.com/compose/overview/>
- Docker Compose file format: <https://docs.docker.com/compose/compose-file/>

NGINX load balancing with Docker

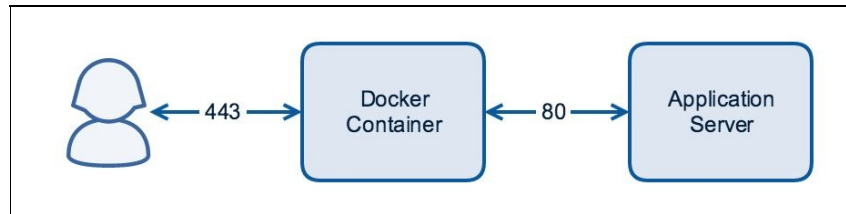
Once you've tackled the conversion from complex deploy scripts into neat `Dockerfile`, the next step is to deal with scale. As for most other problems, Docker has a solution for this too; scaling is one of those.

While newer versions of Docker have added native load balancing, it's still quite simplistic. This makes NGINX a better choice for many uses. Since you're already familiar with the workings of NGINX, it's easily adapted to provide load balancing within a Docker environment.

In a larger deployment, we'd use a more formal service discovery tool, such as `consul` or `etcd`, in order to provide more granular control. This recipe will simply use the built-in DNS capability of Docker in order to round-robin the requests.

Getting ready

This recipe has been simplified to run on a single VM. In most real-world scenarios, this would be spread across multiple VMs. Here's what our test scenario looks like:



We're using the HTest tool, as covered back in [Chapter 8, Load Balancing](#), as the container we want to scale and load balance.

How to do it...

To deploy our load-balanced platform, first we're going to create an NGINX container. Here's our `default.conf`:

```
server {
    listen          443 ssl http2;
    server_name     dockerdemo.nginxcookbook.com;

    ssl_certificate  /etc/ssl/dockerdemo.crt;
    ssl_certificate_key /etc/ssl/dockerdemo.key;
    ssl_dhparam      /etc/ssl/dh4096.pem;
    ssl_protocols    TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers      HIGH:!aNULL:!MD5;

    ssl_prefer_server_ciphers on;
    client_max_body_size 75M;

    location / {
        resolver 127.0.0.11 valid=1;
        set $backend "http://htestbackend:8000";
        proxy_pass $backend;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
    }
}
```

Additional files, such as the associated SSL certificates and `Dockerfile`, are also required.



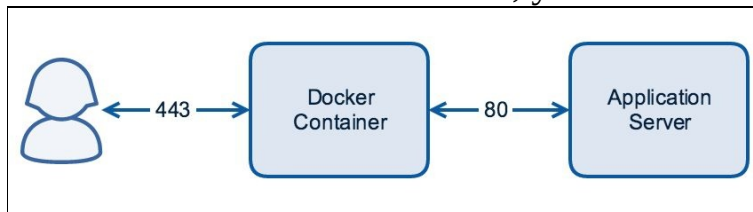
These are also available directly from GitHub at <https://github.com/timbutler/nginxcookbook>.

As we're still using Docker Compose for the deployment, we'll also need our updated `docker-compose.yml` configuration: version: '3' networks: htestnet: services: nginx-proxy: build: ./ ports: - "443:443" networks: - htestnet htest: image: timbutler/htest networks: htestnet: aliases: - htestbackend

Again, this is quite similar to our previous recipe, as the structure of what we're trying to achieve is quite similar. Once we have the configuration, we can now build and start our containers: **docker-compose up -d**

This will start both the `nginx` container as well as `htest`; it will also start the

associated network. If successful, you should see an output like this:



By default, however, there's only one instance of `htest` running. In order to scale this, we can simply tell Docker Compose how many instances we want to run. Here's an example of this: **`docker-compose scale htest=4`**

This will start an additional three `htest` containers:

```
Sending build context to Docker daemon 9.728 kB
Step 1/8 : FROM nginx:latest
--> 6b914bbcb89e
Step 2/8 : MAINTAINER Tim Butler "info@nginxcookbook.com"
--> Using cache
--> 8a6de11eeb87
Step 3/8 : COPY default.conf /etc/nginx/conf.d/
--> Using cache
--> 2c2237b03177
Step 4/8 : COPY dockerdemo.crt /etc/ssl/dockerdemo.crt
--> c5c10b0c4f0a
Removing intermediate container 5a7604d8fb31
Step 5/8 : COPY dockerdemo.key /etc/ssl/dockerdemo.key
--> dacfd8a06576
Removing intermediate container 57c8ff6ba2c5
Step 6/8 : COPY dh4096.pem /etc/ssl/dh4096.pem
--> 655844f5d05b
Removing intermediate container dd6ec56f6eaa
Step 7/8 : EXPOSE 443
--> Running in 189430ecadb9
--> 7893d538042a
Removing intermediate container 189430ecadb9
Step 8/8 : CMD nginx -g daemon off;
--> Running in 65f32579fb70
--> b4007604b773
Removing intermediate container 65f32579fb70
Successfully built b4007604b773
```

Because we have NGINX set to call the proxy backend by its hostname, these are now called in a basic round-robin to each of the containers. If you browse to the site, you should see the counter for `htest` jumping around as each instance serves a request.

How it works...

In our NGINX configuration (`default.conf`), we add the resolver directive (`resolver 127.0.0.11 valid=1`) that directs NGINX to use the built-in Docker DNS resolver. In order to distribute the load, setting the validity to 1 means any TTL is ignored and the result only stays valid for 1 second.

Then, we set the variable `$backend` to `http://htestbackend:8000`. The use of a variable ensures that it's evaluated each time in order to have the IP address updated.

Within `Dockerfile`, we have set the alias for the `htest` container to `htestbackend`. Once we call the `scale` command, this starts three additional containers. Although a unique name is allocated to each container (for example, `loadbalancer_htest_1`), the alias ensures that there's a common name. This avoids having to rewrite the configuration of NGINX each time a new system is added, giving us the ability to add additional backends without reconfiguring them.

See also

- NGINX `resolver` directive: http://nginx.org/en/docs/http/ngx_http_core_module.html#resolver
- To know more about the `docker compose scale` command: <https://docs.docker.com/compose/reference/scale/>

Performance Tuning

In this chapter, we will cover the following recipes:

- Gzipping content in NGINX
- Enhancing NGINX with keep alive
- Tuning worker processes and connections
- Fine tuning basic Linux system limits
- Integrating `ngx_pagespeed`

Introduction

Once you have your NGINX configuration working, you can turn your focus to fine tuning to enhance performance. A few sections of this chapter will focus on delivering increased performance for users, while others will focus on delivering performance enhancements at a server level to allow greater concurrency.

As with any tuning, you need to ensure that you understand the limits first.

Premature optimization is the root of all evil.

– Donald Knuth, 1974

In the context of NGINX, you need to ensure that you know what the limits are before changing them. Not all changes will necessarily result in performance increases if they don't suit your system or if they're not a current limitation.

On the flip side, optimization of your NGINX server is also critical to ensure that your website or application can handle increased traffic and to ensure fast responses. Especially when it comes to e-commerce platforms, keeping user engagement through low response times is paramount. Studies conducted by Amazon found that an increase in the page load time by one second would result in a loss of over \$1.6 billion in revenue each year. Even without e-commerce, the last thing that you would want a user to be doing is waiting unnecessarily, as they will disengage very quickly.

Gzipping content in NGINX

Gzip is a compression format, which is based on the DEFLATE algorithm and commonly found in most Unix environments. Compressing your HTML files is an easy way to reduce the amount of data transferred from NGINX to the browser. This in turn means that pages also load quicker as the file can be transferred in a shorter time due to the reduced size.

While it usually shows the most gain, HTML-based content isn't the only thing, which can compress easily. Any text-based file (for example, JavaScript or CSS) will generally compress by 70 percent or more, which can be quite significant with modern websites.

Of course, enabling compression isn't free. There is a performance hit in server load, as the server needs to use CPU cycles to compress the data. While this used to be a large consideration, with modern CPU's, the performance hit is far outweighed by the benefit of the compression.

Getting ready

The NGINX `gzip` module is part of the core modules, so no additional installation is required.

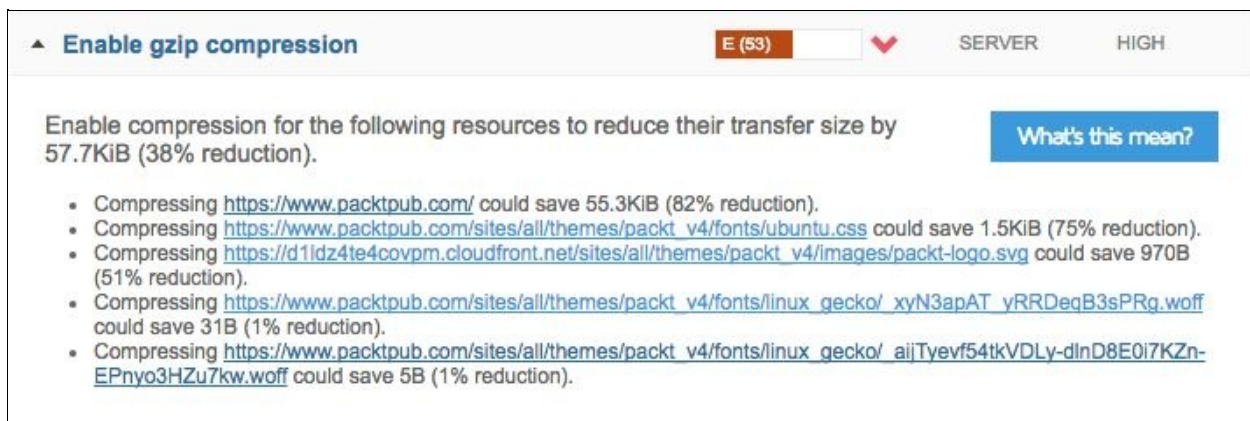
How to do it...

In order to enable `gzip` within NGINX, we need to enable the `gzip` module and explicitly tell it what files to compress. The easiest way to do this server-wide is to create a `gzip.conf` file within the `/etc/nginx/conf.d` directory directly, alongside your server directive files. This could also be set per site or even per location if required; the `gzip` directives can be nested within an existing block directive.

Here's what is required:

```
gzip            on;  
gzip_disable    "MSIE [1-6]\.(?!.*SV1)";  
gzip_proxied    any;  
gzip_types      text/plain text/css application/x-javascript application/javascript text/x  
gzip_vary       on;
```

If you want to measure how much difference *gipping* your files may make, tools such as GTmetrix can outline the reduction in file transmission size. For example, if we look at the <https://www.packtpub.com/> website, we see the following in the Gzip section:



The screenshot shows the 'Gzip' section of a GTmetrix report. At the top, there is a toggle switch labeled 'Enable gzip compression' which is currently turned off. To the right of the toggle are labels 'E (53)', a red heart icon, 'SERVER', and 'HIGH'. Below the toggle, a message states: 'Enable compression for the following resources to reduce their transfer size by 57.7KiB (38% reduction)'. To the right of this message is a blue button labeled 'What's this mean?'. Below the message is a list of five resources that can be compressed, each with a link to the resource and the potential savings in size and percentage reduction.

Resource	Size Reduction
https://www.packtpub.com/	55.3KiB (82% reduction)
https://www.packtpub.com/sites/all/themes/packt_v4/fonts/ubuntu.css	1.5KiB (75% reduction)
https://d1ldz4te4covpm.cloudfront.net/sites/all/themes/packt_v4/images/packt-logo.svg	970B (51% reduction)
https://www.packtpub.com/sites/all/themes/packt_v4/fonts/linux_gecko/ xyN3apAT yRRDeqB3sPRg.woff	31B (1% reduction)
https://www.packtpub.com/sites/all/themes/packt_v4/fonts/linux_gecko/ aijTyevf54tkVDLy-dlnD8E0i7KZn-EPnyo3HZu7kw.woff	5B (1% reduction)

While the savings in this example aren't massive, the 82 percent reduction can show you what's possible for other sites. If there were other files, such as JS or CSS, which weren't already compressed, the decrease becomes much more significant.

How it works...

The first part of our configuration explicitly turns the `gzip` module on. Then, to maintain compatibility with really old versions of Internet Explorer (which hopefully nobody still uses), we disable `gzip` using the `MSIE [1-6]\.(?!.*SV1)` regex.

Then, `gzip_proxied` sets which proxied connections will use `gzip`, which we set to `any` to cover all requests. `gzip_types` is then used to set what file types are to be compressed. This is matched with the MIME type, for example, `text/plain`. We explicitly set types, as not every file type can be compressed further (for example, JPEG images).

Lastly, we set `gzip_vary` to `on`. This sets the `Vary: Accept-Encoding` header, which specifies that both **Content Distribution Networks (CDN)** and upstream proxies store a copy of the file as both compressed and uncompressed. While every modern browser supports Gzip compression, there are still some minor browsers and script-based HTTP tools which don't. Instructing the upstream CDN or proxy to store both shows that they're still able to support these older systems.

There's more...

If the performance hit from gzipping the files on the fly is too much, NGINX also allows the ability to precompress the files to serve. While this means that there's a bit of extra maintenance work required, this can be incorporated into an existing build process (such as **Grunt** or **Gulp**) to reduce the steps required.

To enable in NGINX, we modify our `gzip.conf` file to look like the following code:

```
gzip            on;  
gzip_static on;  
gzip_disable "MSIE [1-6]\.(?!.*SV1)";  
gzip_proxied any;  
gzip_types     text/plain text/css application/x-javascript application/javascript text/x  
gzip_vary      on;
```

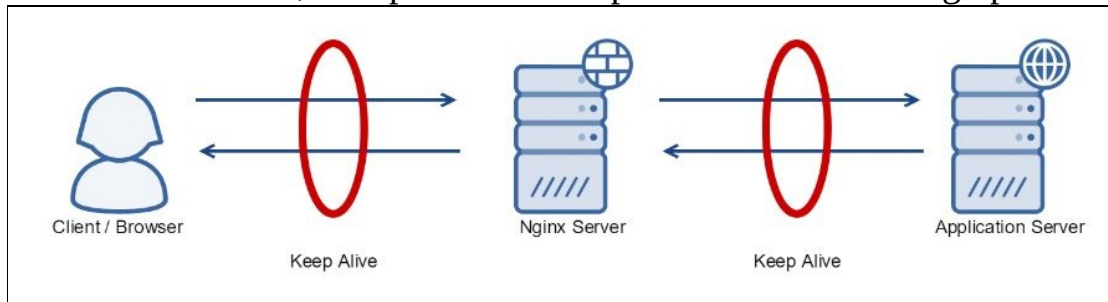
With `gzip_static` set to `on`, NGINX will serve the precompressed files if they exist.

See also

- The Gzip NGINX module can be found at http://nginx.org/en/docs/http/nginx_http_gzip_module.html
- Refer to GTmetrix's official website at <https://gtmetrix.com/>

Enhancing NGINX with keep alive

Using a persistent HTTP connection between the server and the browser speeds up additional requests, as there's no extra handshaking required. Especially over more latent connections, this can increase the overall performance. If NGINX is being used as a reverse proxy (as detailed in [Chapter 7, Reverse Proxy](#)), it's also important to ensure that these connections have `keepalive` enabled to ensure high throughput while minimizing latency. The following diagram highlights both areas where the `keepalive` packets are important to maintain high performance:



This persistent connection remains established using **Keep Alive** packets, so that the connections remain open for minutes rather than closing once they are complete. This reuse can be immediate for additional CSS/JS files or as further pages and resources are requested.

While some of the client-side gains are negated using HTTP/2 (which multiplexes connections as well as uses `keepalive`), it's still necessary for HTTP (non-SSL) connections and upstream connections.

Getting ready

The NGINX `keepalive` module is part of the core modules, so no additional installation is required.

```
server {  
  
    listen 80; server_name express.nginxcookbook.com;  
  
    access_log /var/log/nginx/express-access.log combined;  
  
    location / {  
  
        proxy_pass http://127.0.0.1:3000; proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade; proxy_set_header  
        Connection "upgrade"; keepalive 8; }  
  
    }
```

How it works...

By adding the `keepalive` directive, we define the maximum number of idle connections to keep open using keep alives. In our recipe, we specified a maximum of eight idle connections. It's important to note that this isn't the maximum number of connections in total, this only defines the number of idle connections to keep open.

We can confirm that `keepalive` is working by checking the connections from the NGINX server. To do this, we use the `ss` command with the `-o` flag to display timer information in relation to the socket. For example, we can run the following command:

```
| ss -tpno
```

With our Express-based demo, you should see something like the following:

```
| State Recv-Q Send-Q Local Address:Port Peer Address:Port
| ESTAB 0      0      127.0.0.1:3000      127.0.0.1:33396
| users: (("nodejs",pid=4669,fd=11))
| ESTAB 0      0      xx.xx.xx.xx:80      yy.yy.yy.yy:51239
| users: (("nginx",pid=4705,fd=29)) timer:(keepalive,3min44sec,0)
| ESTAB 0      0      127.0.0.1:33396      127.0.0.1:3000 users: (("nginx",pid=4705,fd=34))
```

We can see that sockets which have a `keepalive` packet have been flagged with a timer output to show the expiry.

If you need to test if the browser is seeing the `keepalive` response from the server, you can do this with browser developer tools such as Chrome **Developer Tools (DevTools)**. In your browser of preference, open the developer tools and look for the response headers:

```
Accept: text/html,application/xhtml+xml,application/xml;q=
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-AU,en-GB;q=0.8,en-US;q=0.6,en;q=0.4
Cache-Control: no-cache
Connection: keep-alive
Host: nginx.org
Pragma: no-cache
```

In the preceding screenshot, we can see that the server responded with `Connection:`

keep-alive. This means that the keepalive packets are supported and working.

See also

The NGINX `keepalive` documentation can be found at http://nginx.org/en/docs/http/nginx_http_upstream_module.html#keepalive

Tuning worker processes and connections

One of the first limits you'll find with running NGINX at scale is the defaults for the worker processes and connections. At a low level, an NGINX worker process is the dedicated event handler for processing all requests.

The defaults for most NGINX installations are 512 worker connections and 1 worker process. While these defaults work in most scenarios, a very busy server can benefit from adjusting these levels to suit your environment. There is no one-size-fits-all scenario when it comes to the correct values, so it's important to know where you're hitting limits and therefore, how to adjust to overcome them.

Setting the limits too high can result in increased memory and CPU overhead, which would have the overall effect of reduced performance rather than increasing it. Thankfully, NGINX will log when it hits certain limits, which is why the logging (as covered in [Chapter 5, Logging](#)) and metrics of your systems are paramount to maintaining high performance.

Getting ready

No special requirements exist for modifying worker process or connection directives.

How to do it...

As the worker processes and worker connections can be independently adjusted, we can adjust either or both of them; depending on the limits which you're hitting.

Worker processes

To adjust the number of worker processes, we need to edit the main NGINX configuration file. For most installations, this will be located at `/etc/nginx/nginx.conf`, and the directive is generally the first line in the file. Here's what you may see for a default value: `worker_processes 1;`

If the server is dedicated to just running NGINX (for example, it doesn't have the database and other services also running on it), a good rule of thumb is to set it to the number of CPU's available. Consider this example when you have four CPU's: `worker_processes 4;`

If you have a high number of connections and they're not CPU bound (for example, heavy disk I/O), having more processes than CPU's may assist with increasing the overall throughput of the server.

Lastly, NGINX can attempt to autodetect the number of CPU's in your system by setting the value to auto: `worker_processes auto;`

If you're not sure what value to use, auto is the best option to use.

```
events {  
  
    worker_connections 4096;  
  
}
```

As increasing the maximum number of connections means that additional system resources are required, caution should be exercised when making a change. If the server hits the limit for `worker_connections`, this will be logged in the NGINX error log. By ensuring that this is monitored alongside server resources, you can ensure that it has been set to the correct limit.

There's more...

The worker connections can be further enhanced on modern systems by a few extra directives. Here's what our updated block directive looks like:

```
events {  
    worker_connections 4096;  
    multi_accept on;  
    use epoll;  
}
```

We have added the additional `multi_accept` directive and set it to `on`. This tells the NGINX worker to accept more than one connection at once when there are a high number of new, incoming connections.

Then, we set the `use` directive to `epoll`. This is the method NGINX uses to process the connections. While on every modern system this should be automatically set to `epoll` by default, we can explicitly set this to ensure that it's used.

See also

- For more information on worker processes, refer to http://nginx.org/en/docs/nginx_core_module.html#worker_processes
- For more information on worker connections, refer to http://nginx.org/en/docs/nginx_core_module.html#worker_connections

Fine tuning basic Linux system limits

The most popular hosting **Operating System (OS)** for NGINX is Linux, which is why we have focused on it in this book. Like NGINX, the out-of-the-box parameters are a good balance between resource usage and performance.

One of the neat features of Linux is the fact that most of the kernel (the *engine* of the OS) can be tweaked and tuned as required. In fact, nearly every underlying aspect can be adjusted quite easily to ensure that you can perfectly tune it to your needs. With over 1,000 configurable parameters in the kernel, there's virtually infinite tuning available.

Like our warning at the beginning of this chapter however, larger numbers don't necessarily reflect greater performance. It's important to ensure that you understand what parameters you're changing to understand the impact they'll have.

Getting ready

Make sure that you have a backup of your existing server before making any changes. As the changing of the kernel parameters could result in adverse performance or even an unworkable system, it's advisable to perform this on a development or staging system first.

How to do it...

Each kernel parameter can be set in real time, which allows you to test and refine the changes on the fly. To do this, you can use the `sysctl` program to change these parameters. For starters, we can ensure that TCP syncookies (a method of resisting low level denial of service attacks) are enabled:

```
| sysctl -w net.ipv4.tcp_syncookies=1
```

If you want the change to be persistent between boots, we can add this to the `sysctl` configuration file, generally located at `/etc/sysctl.conf`. Here's what the configuration line should look like:

```
| net.ipv4.tcp_syncookies = 1
```

You can test and retrieve what value any of the kernel parameters are set to using `sysctl` again in read-only mode:

```
| sysctl net.ipv4.tcp_syncookies
```

If you've set it correctly, you should see the following result:

```
| net.ipv4.tcp_syncookies = 1
```

When you're trying to determine if limits have been hit, ensure that you check the kernel ring buffer for errors by running the `dmesg` utility. This logs the output from any kernel module, which generally occurs when they either encounter a limit or error and usually the first port of call to determine what limits you've hit.

If you have a busy server, one of the first Linux kernel limits you may find yourself hitting is when there are some delays in processing and you still have a large number of incoming connections which haven't yet been accepted. While there is a buffer, once you hit this buffer, the server will simply drop any further incoming connections which will cause disruption. To increase this limit, we can adjust the limit by increasing the value for `net.core.somaxconn`. On many systems, this defaults to 128 connections, but we can increase this by running the following command:

```
| sysctl-w net.core.somaxconn=1024
```

With a large amount of incoming connections, you may also find yourself running out of ephemeral ports for newer connections. As one of the final stages of a TCP connection is the `TIME_WAIT` stage, here the connection has been requested to be closed but it's held open just in case there are any further packets. On a busy server, this can result in thousands of connections being held in a `TIME_WAIT` state and, by default, these need to be completely closed before they can be reused. We can see the state of the TCP ports on a server by running the following command:

```
| ss -ant | awk '{print $1}' | sort | uniq -c | sort -n
```

Here's the output from a moderately-low used server:

```
| 1 CLOSING
| 1 State
| 2 CLOSE-WAIT
| 3 LAST-ACK
| 3 SYN-RCV
| 5 FIN-WAIT-1
| 59 FIN-WAIT-2
| 1311 LISTEN
| 1516 ESTAB
| 4210 TIME-WAIT
```

If the server becomes very busy, it's possible that all the ports available will be locked in the `TIME_WAIT` state. There are two approaches to overcoming this limitation. The first is to reduce the time we hold a connection in the `TIME_WAIT` stage. This can be done by lowering the default of 60 seconds to 10 seconds:

```
| sysctl -w net.ipv4.tcp_fin_timeout=10
```

Secondly, we could simply tell the Linux kernel to reuse ports still in the `TIME_WAIT` stage for new connections, if required:

```
| sysctl -w net.ipv4.tcp_tw_reuse=1
```

This generally isn't enabled by default due to possible conflicts and issues with old, legacy applications, but should be safe to enable for an NGINX server.

You may also find many blogs and articles advising you to increase the buffer sizes for TCP, but these generally focus on increasing the buffer sizes for file serving. Unless you're using NGINX to serve large files, the default values are generally high enough for low latency connections.

See also

- The NGINX tuning blog can be found at <https://www.nginx.com/blog/tuning-nginx/>
- For more information on kernel tuning, refer to <https://www.linux.com/news/kernel-tuning-sysctl>

Integrating ngx_pagespeed

As the kings of high performance, Google has given us many tools and enhancements that have benefited the web world enormously. Google Chrome (as of 2017) has over 60 percent of the browser market share and its drive for performance has forced other browsers to play catch-up.

Not to be outdone at the server level, Google also has vested interest in ensuring that websites are highly performant as well. This is because faster sites offer a better user experience, which is important when you're trying to offer highly relevant search results. To expedite this, Google released `ngx_pagespeed`, which is a module for NGINX that tries to apply Google's best practices to reduce both latency and bandwidth for websites.

While many (if not all) of these optimizations can be manually applied, or should be part of any highly-performant development workflow, not everyone has the time to put the same amount of focus on overall website performance. This is especially common with smaller business websites, where the development has been outsourced to a third party, but don't have the budget to fully optimize.

Getting ready

The `ngx_pagespeed` module requires you to compile the module and NGINX from source, which can be achieved in two ways. The first is to use the automated script provided by Google:

```
| bash <(curl -f -L -sS https://ngxpagespeed.com/install) \  
| --nginx-version latest
```



Manually inspecting any script before running it is advisable, especially when downloading from new or unknown sources.

This script will install any required dependencies, then download the latest mainline NGINX edition, and then add the `ngx_pagespeed` module. This script isn't completely headless and may ask you to confirm some basic parameters such as additional modules to compile (if required). If you intend to use this on a production server, you will need to adjust some of the installation paths to suit your environment.

The second method is via a manual installation, which can also be used if you need to modify the standard build. Details of the manual installation steps are located on the `ngx_pagespeed` website. With the advent of dynamic modules within NGINX, there should hopefully be a compiled binary version available very soon as well.

How to do it...

To test the differences with and without the module enabled, I've used a typical Bootstrap-based website which incorporates several JavaScript and **Cascading Style Sheets (CSS)** scripts:

To serve these files, we have a basic NGINX configuration:

```
server {
    listen      80;
    server_name pagespeed.nginxcookbook.com;

    access_log  /var/log/nginx/test-access.log combined;

    location / {
        root    /var/www/test;
        index   index.html;
    }
}
```

With a basic site set up, we can now enable the `nginx_pagespeed` module. Before we enable the module, we first need to create a directory for cache file storage. We can do this using the following command:

```
| mkdir /var/ngx_pagespeed-cache
```

We also need to ensure that it's writable by NGINX, so we simply set the ownership to be the `nginx` user, using the following command:

```
| chown nginx:nginx /var/ngx_pagespeed-cache
```

With the cache directory ready, we can now load the module by adding the following lines into the server block directive:

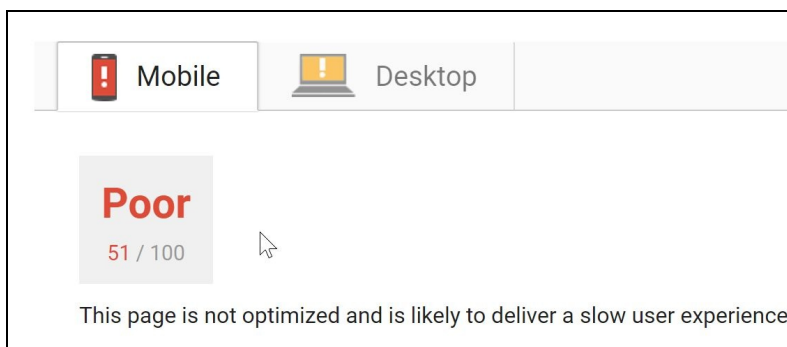
```
| pagespeed on;
| pagespeed FileCachePath /var/ngx_pagespeed-cache;
```

While it may seem overly simple, there's an enormous amount of complexity and work which goes on in the module. Since some of the options may cause issues with a small number of sites, there's also the ability to disable certain sub-modules. For instance, if we wanted to disable combining CSS files, we can disable the filter by adding the following directive:

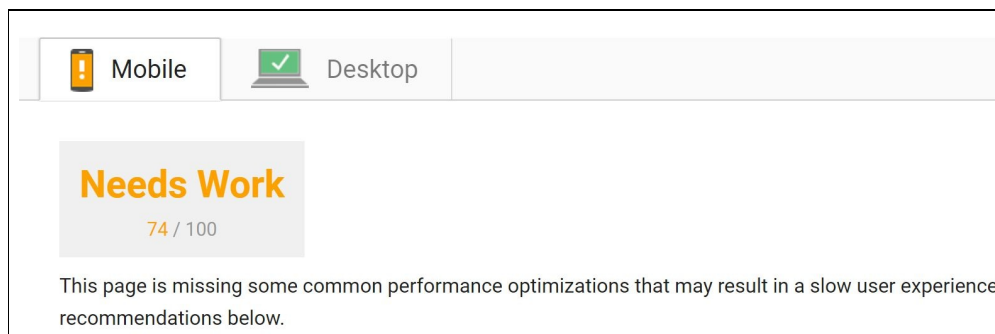
```
| pagespeed DisableFilters rewrite_images,combine_css;
```

How it works...

Using Google's **PageSpeed Insights**, we can see the out-of-the-box score for the website without any optimization enabled:



Obviously, 51/100 isn't a great score; this is due to multiple CSS and JS files that haven't been minified, are not compressed, and use no explicit browser caching. With `ngx_pagespeed` enabled, we get a much better result:



This has instantly given the website a boost in performance, but the simple score doesn't tell the whole story. When comparing the differences in the number of requests, the total has nearly halved.

	Without <code>ngx_pagespeed</code>	With <code>ngx_pagespeed</code>
Files Loaded	18	11

Total Transferred	540 kB	214 kB
--------------------------	--------	--------

While this is only for a very basic site, as the improvements show, there are significant performance gains for nearly zero effort.

Like many of the systems that work, Google's optimizations have quite a number of neat features. As compressing and minifying CSS/JS can be CPU intensive, on the first page load (without the cache being warmed), NGINX will simply serve the site in the original format. In the background, the module queues these tasks and, once available, they will be served directly from the cache.

There's more...

If you want to see what's going on, we can enable the admin area for `mod_pagespeed`. To do this, we need to add configuration items outside of the main server directive. Here's the code to add:

```
pagespeed on;  
pagespeed FileCachePath /var/nginx_pagespeed-cache;  
pagespeed statistics on;  
pagespeed StatisticsLogging on;  
pagespeed LogDir /var/log/pagespeed;  
pagespeed MessageBufferSize 100000;
```

This allows us to see what's going on within the module, including details such as hits and misses from the cache, image compression, CSS, and JavaScript minification, and more:

Pagespeed Admin

[Statistics](#)

[Configuration \(SPDY\)](#)

[Histograms](#)

[Caches](#)


[Console](#)

[Message History](#)

Auto refresh (every 5 seconds): ☐

Filter:

T

Name 	Value	Number of c...
cache_backend_hits	35	1
cache_backend_misses	14	1
cache_batcher_dropped_gets	0	0
cache_deletes	0	0
cache_expirations	0	0
cache_extensions	0	0
cache_fallbacks	0	0
cache_flush_count	0	0
cache_flush_timestamp_ms	0	0
cache_hits	16	1
cache_inserts	23	1
cache_misses	33	1
cache_time_us	116527	1

If this is a production environment, ensure that you restrict access so that it can't be used for malicious reasons.

See also

- To learn more about NGINX PageSpeed, refer to <http://ngxpagespeed.com/>
- More information about installation reference is available at <https://modpagespeed.com/doc/build ngx pagespeed from source>
- More information about PageSpeed Insights can be found at <https://developers.google.com/speed/pagespeed/insights/>

OpenResty

In this chapter, we will cover the following topics:

- Installing OpenResty
- Getting started with OpenResty Lua
- Lua microservices with OpenResty
- A simple hit counter with a Redis backend
- Powering API gateways with OpenResty

Introduction

If you've ever wondered whether you can make a few changes to NGINX dynamically or wanted a bit more flexibility, then you're going to love OpenResty.



Think of OpenResty as NGINX with the kitchen sink thrown in; it's a combination of NGINX, along with Lua scripting, and several additional third-party modules all packaged up and ready to use. The inclusion of Lua scripting and additional modules allows NGINX to be extended to be a full web application rather than simply a web server.

Some may fear that this additional functionality comes with a performance hit, but this simply isn't the case. Large platforms, such as **Cloudflare**, use a combination of NGINX and Lua to achieve what they do at scale, and it's due to the power of OpenResty. In fact, the original creator, Yichun Zhang worked for Cloudflare on OpenResty and has now formed a separate OpenResty foundation to steward the platform going forward.

This power allows complicated scenarios such as **Web Application Firewalls (WAFs)** to be tightly integrated at the website level, allowing the combination of per-site flexibility with the high-speed aspects NGINX is known for. In fact, many are even starting to use OpenResty for their full framework and it can be especially effective for a simple microservice-driven system.

This is just one of the many examples. In this chapter, you'll see that OpenResty is a great fit for many scenarios where you want to use NGINX but also wish to add a bit of dynamic flair.

Installing OpenResty

OpenResty is packaged for the easy installation of most Linux distributions, but there are binary packages for both Windows and OS X available as well. As most production deployments will predominantly be Linux-based, we'll concentrate on Linux for our recipe.

Getting ready

If you have NGINX already installed, you'll need to uninstall it first to remove any other conflicts.

How to do it...

Official repositories exist for most major Linux installations, but we'll focus on just CentOS 7 and Ubuntu 16.04 LTS to cover the two most common scenarios.

CentOS

To add the repository, the first thing we need is the `yum-utils` package. This makes the creation of repositories as simple as a one-line installation: **yum install -y yum-utils**

With `yum-utils` installed, we can now create the `openresty` repository on our server:

```
| yum-config-manager --add-repo https://openresty.org/package/centos/openresty.repo
```

This will automatically fetch the remote repository file and place it in the correct location for you.

With the repository installed and enabled, we can now install OpenResty:

```
| yum install -y openresty
```

This will install the latest OpenResty package, as well as all the required dependencies.

To enable the service to start on boot, we can enable it via `systemd`:

```
| systemctl enable openresty
```

You can start the service via `systemd` as well:

```
| systemctl start openresty
```

This will start NGINX, which will be preconfigured with all the OpenResty additions. As it comes with a simple configuration (as NGINX does out-of-the-box), you can quickly open a browser and see the output via the IP to confirm it's

Welcome to OpenResty!

If you see this page, the OpenResty web platform is successfully installed and working. Further configuration is required.

For online documentation and support please refer to openresty.org.

Thank you for flying OpenResty.

working:

Ubuntu

To install OpenResty on an Ubuntu-based system, first we need to import the **GPG** key used for signing the packages:

```
| wget -qO - https://openresty.org/package/pubkey.gpg | apt-key add -
```

Like the CentOS installation, we can use a helper package to make the installation of the repository easy. To install it, use the following command:

```
| apt install -y software-properties-common
```

We can now install the repository for OpenResty and then refresh the package indexes:

```
| add-apt-repository -y "deb http://openresty.org/package/ubuntu $(lsb_release -sc) main"
| apt update
```

With the repository installed, we can now install the OpenResty package:

```
| apt install -y openresty
```

Once all the dependencies and OpenResty packages are installed, you can now set the service to start on boot and then start it so that you can test the service. You can do this via `systemd`: **systemctl enable openresty systemctl start openresty**

If you don't see any errors, you will then be able to browse to the IP address of your server (or virtual machine) and see the OpenResty test page, as shown in the previous CentOS installation instructions.

How it works...

The locations of the configuration files and system libraries are slightly different to a standard NGINX installation, so it's important to remember the location. By default, OpenResty installs to `/usr/local/openresty`. For example, if you look for the NGINX configuration files, they'll be stored in `/usr/local/openresty/nginx/conf`.

If you look at the standard NGINX configuration file, you won't see any real difference between it and the standard package. Because OpenResty is essentially just NGINX with additional modules compiled, you can easily take an existing NGINX configuration (like any of the ones covered in this book) and extend the functionality with the additional OpenResty modules.

We can confirm these modules are available by running the following command:

```
| /usr/local/openresty/bin/openresty -v
```

This should give you an output similar to the following:

```
nginx version: openresty/1.11.2.3
built with OpenSSL 1.0.2k 26 Jan 2017
TLS SNI support enabled
configure arguments: --prefix=/usr/local/openresty/nginx --with-cc-opt='-O2 -I/usr/loca
```

We can confirm that there are additional modules installed, such as `lua_upstream` and `lua_jit`, which form the core of the OpenResty capabilities.

See also

For official installation instructions, refer to <https://openresty.org/en/installation.html>.

Getting started with OpenResty Lua

One of the key powers of OpenResty is the built-in Lua scripting language. For those not familiar with Lua, it's a high-performance, yet lightweight scripting language. This is why, when it's combined with the NGINX event engine, it results in a very powerful combination.

Being a dynamically typed and interpreted language makes Lua similar to other scripting languages, such as JavaScript, but there are some subtle differences (especially syntax-wise). If you're new to Lua, then it's worthwhile reading through a few basic tutorials to familiarize yourself with the syntax and differences.

Getting ready

For this recipe, we'll use the standard OpenResty modules, so no further changes are required to get started.

How to do it...

We'll start with using one of the most basic functions in OpenResty and Lua, which is the `content_by_lua_block` block directive. This allows us to insert Lua code directly in line with our NGINX configuration, providing rapid and dynamic changes. The first recipe returns a basic string:

```
location /simpletest {
    default_type 'text/plain';
    content_by_lua_block {
        ngx.say('This is a simple test!')
    }
}
```

If you browse to the URL (or use cURL to make the request), you should simply get `This is a simple test` as the HTTP response. Running a simple Apache Benchmark against this URL (just to show a baseline performance) shows that it can serve this URL over 20,000 times a second on a modestly resourced VM. While the code isn't overly complex, it does show that the overheads for adding Lua have a very minimal effect on performance.

If you need to return the data as JSON, then this is quite simple to do as well. Using the basic example, as we used previously, we can leverage the Lua CJSON library (compiled with OpenResty by default) to encode the output:

```
location /simplejsontest {
    default_type 'application/json';
    content_by_lua_block {
        local cjson = require "cjson.safe"
        ngx.say(cjson.encode({test="Encoded with CJSON",enabled=true}))
    }
}
```

If we call the `/simplejsontest` URL, you should see the following output:

```
{"test":"Encoded with CJSON","enabled":true}
```

This, of course, barely scratches the surface of what can be achieved with Lua, but this should at least get you started.

How it works...

In both recipes, we utilized Lua modules to provide (albeit simplistic) output based on Lua code. While the code is in line with the NGINX configuration, it runs directly within each worker process. This gives it massive concurrency out-of-the-box, which, combined with the native speed of Lua, means it's incredibly powerful.

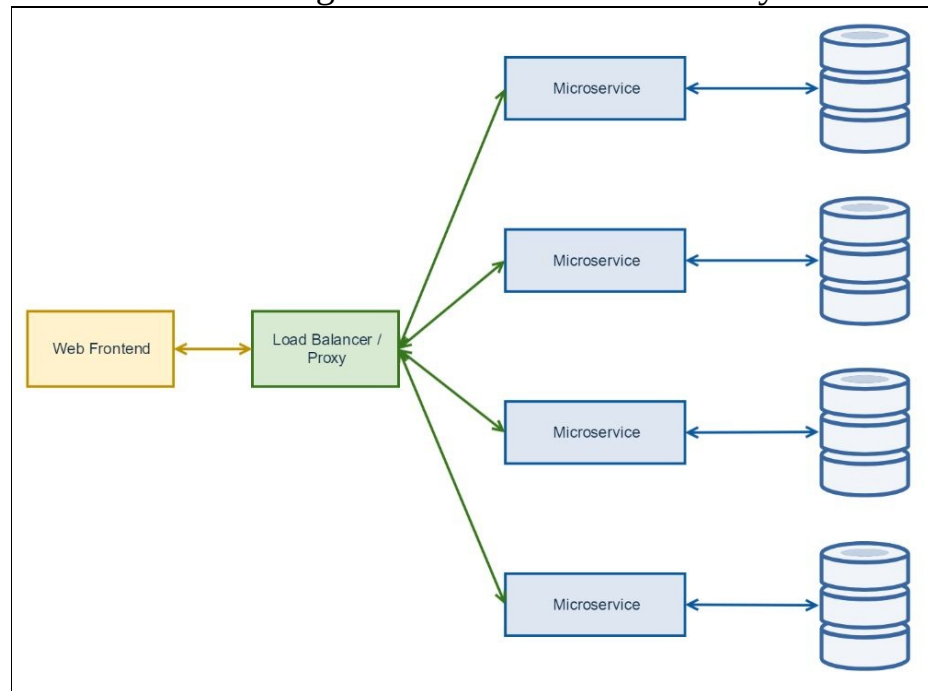
See also

For more details on the Lua module, refer to <https://github.com/openresty/lua-nginx-module#readme>.

Lua microservices with OpenResty

One of the quickest, natural extensions of OpenResty is to create and run a microservice directly, rather than having to proxy it to another external service. For those not familiar with microservices, this is a methodology of breaking down a software platform into small, independent services rather than a single, monolithic system. Here's a basic diagram of how the services may look for a

web application:



This means that each microservice can be independently upgraded, changed, and scaled as required; keeping it to one task means the code should remain easier to manage.

Getting ready

In this recipe, we're going to focus on just one microservice. In a real-world deployment, these microservices could be as high as 150 for a complex platform, and many typically hover between the range of 10-30.

For this microservice, we're going to take advantage of the built-in Lua DNS module (`lua-resty-dns`) to provide a resolution tool and return the result as JSON. As a real-world example, we're going to look up the **Mail Exchanger (MX)** record. This could be part of a platform for email migration, anti-spam validation, or similar and would traditionally require NGINX to proxy the connection to an external application.

```

location /getmxrecords {

    default_type 'application/json'; content_by_lua_block {

        local cJSON = require "cjson.safe"

        local resolver = require "resty.dns.resolver"

        local r, err = resolver:new{

            nameservers = {"8.8.8.8"}

        }

        if not r then

            ngx.say(cJSON.encode({result="failed", <br/> message="Failed to
initiate the resolver. <br/> Reason: "..err}))) return

        end


        local domain = ngx.var.arg_domain if not domain or not
string.match(domain, "[%w]*[%.]?[%w]*") then
ngx.say(cJSON.encode({result="failed", <br/> message="Invalid
domain entered"}))) return

        end


        local result, err = r:query(domain, { qtype = r.TYPE_MX }) if not
result then

            ngx.say(cJSON.encode({result="failed", <br/> message="Failed to

```

```
return a result.<br/> Reason: "..err})) return
```

```
end
```

```
ngx.say(cjson.encode({result="success", records=result})) }
```

```
}
```

```
<strong>{"records":[{"exchange":"mxa-  
00082601.gslb.pphosted.com","preference":10,"class":1,"ttl":299,"nan  
{"exchange":"mxb-  
00082601.gslb.pphosted.com","preference":10,"class":1,"ttl":299,"nan  
</strong>
```

There can be more than one MX record; this is why you see an array returned within the JSON data for the records.

How it works...

We start by loading the CJSON and Lua OpenResty DNS modules and initiating the DNS module by setting the nameservers to 8.8.8.8 (Google's free open resolver).

Then, we parse the `GET` argument, named `domain`. Through the NGINX API, Lua can call this directly via the `domain` name. If the get variable you wanted was named `shop`, you could have called it via `ngx.var.arg_shop`.

This is then validated by ensuring the variable is set (for example, the `GET` argument was passed) and then checking for a basic domain. The formats of the regular expressions within Lua are slightly different to the more common **Perl-Compatible Regular Expressions (PCRE)**, but the concept remains the same. We ensure that the domains start with alphanumeric characters (using `%w`); they should contain at least one dot (`.`) and alphanumeric characters. While it's not a perfect validator, the advent of all the new **Top-Level Domains (TLDs)** has made this considerably harder to do.

After ensuring the domain is valid, we run a query, specifying the query type to be `MX`. If we receive a result, this is encoded via the CJSON module to return the results as JSON code.

There's more...

In a production environment, you can use the standard NGINX rate-limiting features (as covered in [Chapter 9](#), *Advanced Features*) to limit the abuse of a service like this, especially if it's exposed directly to the internet. The advantage of OpenResty is that you still have the full power of NGINX to use outside of the enhancements it provides.

See also

- For more information on Lua-resty-dns module, refer to <https://github.com/openresty/lua-resty-dns>
- For more information on Lua NGINX API variables, refer to <https://github.com/openresty/lua-nginx-module#ngxvarvariable>

Simple hit counter with a Redis backend

One simple example to show the ease of extendibility of OpenResty is with a basic hit counter. Taking it a step further, we're going to use a Redis backend so that the counter is both persistent and could also be part of a clustered deployment to give a combined hit counter. This will also introduce you to the basics of how OpenResty can directly talk to many other services outside of the basic proxying of connections or via FPM.

In a deployment where every bit of optimization possible is critical, this could also be used to retrieve cached data direct from Redis, allowing the application servers to simply write cache data to Redis in an asynchronized manner.

Getting ready

We'll need access to a Redis daemon or cluster from this server. This could be in the form of a full cluster or you can simply have Redis installed alongside OpenResty on the same server.

```
location /redistest {  
  
    default_type 'text/plain'; content_by_lua_block {  
  
        local redis = require "resty.redis"  
  
        local red = redis:new() local ok, err = red:connect("127.0.0.1",  
6379) if not ok then ngx.say("Failed to connect to the redis server, the  
error was: ", err) end  
  
        local counter = red:get("counter") if tonumber(counter) == nil then  
counter = 0  
  
        end  
  
        counter = counter + 1  
  
        local ok, err = red:set("counter", counter) ngx.say(counter) }  
}
```

If we call the `/redistest` URL, we should see the counter increase each time the page is refreshed.

How it works...

We again used `content_by_lua_block` to return content, and this contains our basic counter code. It starts by making a connection to Redis on the localhost (`127.0.0.1`) and returns an error if the connection fails.

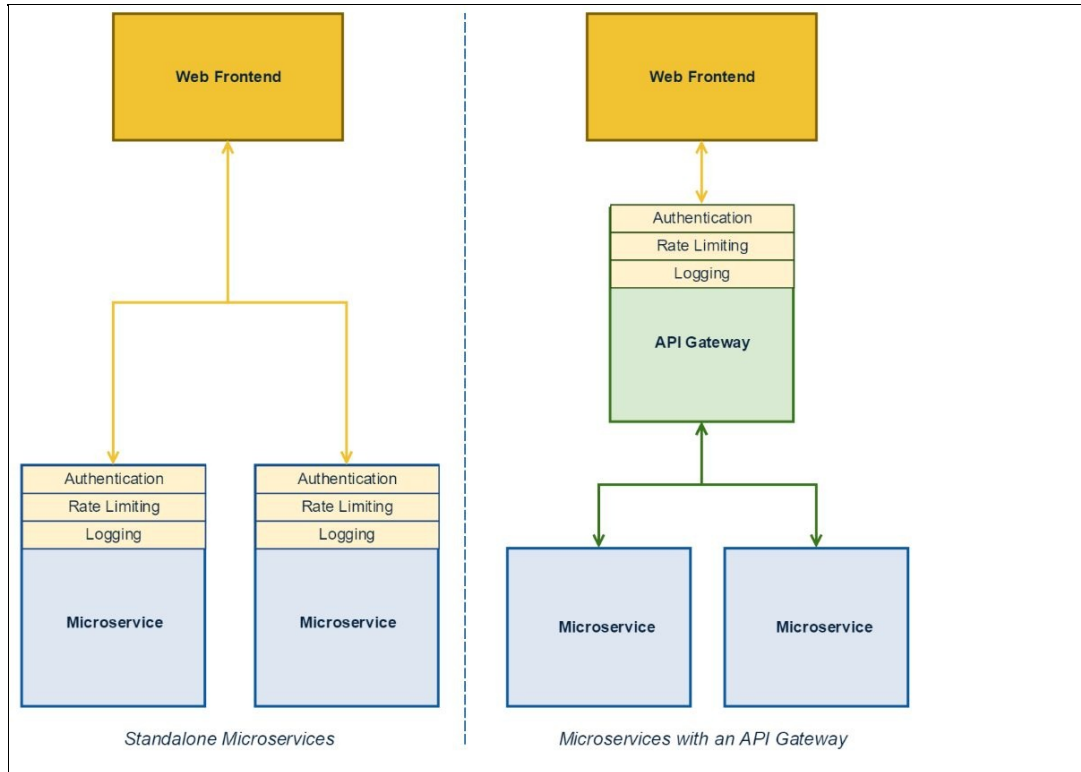
If the connection is successful, we attempt to fetch a value from Redis with a key named `counter`. In the event that there's no data or an invalid number, we set the counter to `0` (zero). Then, we increment the Lua variable to indicate there's been a hit to this URL. This is then stored back in Redis (via the `set` command), and the value of the counter is returned as plain text with a status of `200` (the default for `ngx.say`).

See also

For more information on the `lua-resty-redis` module, refer to <https://github.com/openresty/lua-resty-redis>.

Powering API Gateways with OpenResty

In our previous recipe, we explored a basic microservice to look up DNS records. While this can be limited per service to prevent abuse, ideally, we want to configure a centralized point to manage this. Otherwise, any limits across multiple services will not be considered as a whole and will need to be individually implemented per service. The following figure explains the differences:



To build a centralized API gateway, we need to consider the following points:

- Authentication
- Request routing
- Rate limiting
- Load balancing

- Security
- Logging

This recipe will cover a basic implementation of an API gateway to get you started with some of the core concepts. Because of the ease of implementation, it provides a rapid way to get started with the management of a few, small microservices.

redis-cli SET /api/v1/test http://localhost/simpletest

```
location /proxy/ {  
    rewrite /proxy/(.*) /$1 break; allow 127.0.0.1; deny all; proxy_pass  
    $1; }
```

```
location /api {  
    default_type 'application/json'; access_by_lua_file  
    apigateway/auth.lua; content_by_lua_file apigateway/route.lua; }
```

```
local cJSON = require "cjson.safe"
```

```
local allowedkeys = {"abc123", "def456", "hij789"}
```

```
local function badAuth()
```

```
    ngx.status = 401
```

```
    ngx.say(cJSON.encode({status="error",<br/>  
    errmsg="Authentication Failed"})) ngx.exit(401) end
```

```
local function isAuthorised (key) for index, value in  
    ipairs(allowedkeys) do if value == key then return true end
```

```
    end
```

```
    return false end
```

```
local authKey = ngx.req.get_headers()["X-API-KEY"]
```

```
if authKey == nil then
```

```
    badAuth() elseif not isAuthorised(authKey) then badAuth() end
```

```
local cJSON = require "cjson.safe"
```

```
local redis = require "resty.redis"
```

```
local red = redis:new()
```

```
local ok, err = red:connect("127.0.0.1", 6379) if not ok then
```

```
    ngx.say(cJSON.encode({status="ok", errorMessage=<br/> "Failed to  
connect to the redis server, the error was: "..err}))) ngx.exit(500) end
```

```
local apiroute = red:get(ngx.var.uri) if apiroute == ngx.null then  
ngx.say(cJSON.encode({status="error", errorMessage=<br/> "no  
service at this path"}))) ngx.exit(404) end
```

```
res = ngx.location.capture("/proxy/"..apiroute) if res then
```

```
    ngx.say(cJSON.encode({status="ok", result=res.body}))) else
```

```
    ngx.say(cJSON.encode({status="error", <br/> errorMessage="service  
failed to return a result"}))) ngx.exit(500) end
```

```
<strong>{"status":"error","errmessage":"Authentication Failed"}  
</strong>
```

```
<strong>curl -H "X-API-KEY: abc123"
```

`http://openresty.nginxcookbook.com/api/v1/test`

`{"status":"ok","result":"This is a simple test!\n"}`

How it works...

There's quite a bit going on in this recipe, so we'll go through it section by section. Firstly, the core NGINX configuration has two main sections. The first is the proxy location block directive, which is what we use to make external requests. This is because within `route.lua`, the `location.capture` function only works for internal requests. While, in this recipe, we've only made an internal request, having this `proxy_pass` directive allows us to easily incorporate external calls. It simply takes the remote request as part of the URI to pass through. We've also locked it down (using `allow / deny`) to the localhost to prevent any external misuse.

Next, we define our API location block directive. To keep the code within the main configuration file neat and precise, we store the configuration in an external file so that the code management is easier.

Our `auth.lua` file contains our authentication code. For the sake of keeping this recipe simple to follow, I've created a basic table type (which is an associative array) to store a few test API keys in; for a production system, these would be pulled from an external data source.

We then define a `badAuth` function, which gives us an easy way to return an HTTP 401 error to let the client connection know the connection wasn't authorized.

The next function we've defined is `isAuthorised`. This simply iterates through our table of allowed API keys to determine whether there is a match or not.

Lastly, we extract the `X-API-KEY` header to interrogate the value. If it's nil, that is, the header hasn't been set, we use `badAuth` to return 401. If it's not nil, we use the `isAuthorised` function to determine whether there is a match or not. If there's a match, there's simply nothing further for our code to do and OpenResty will start processing the content components.

This brings us to our routing code contained within the `route.lua` file. Like our previous recipe, we make a connection to our Redis server. This is used to provide dynamic routing. This means, to change our endpoints or even to provide new API functions, there's no requirement to restart our API gateway to

detect these changes.

To get the endpoint URI to call, we use `ngx.var.uri` as the key. In our example, this has been configured as `/api/v1/test`. If this exists, we use `ngx.location.capture` to proxy this through and retrieve the data. A successful return of data is then sent back to the client, parsed as JSON using the `cJSON` module. In the case of an error, we simply return an error message and set the HTTP status to `500`.

There's more...

What's missing from this recipe is any form of **rate limiting**. We could again incorporate either the standard NGINX module or use the `lua-resty-limit-traffic` module to provide extended functionality; alternatively, you can go for a fully featured API system, such as Kong:



Based on OpenResty, Kong offers a very highly configurable API gateway and microservice management system with features such as a REST-based administration, easy horizontal scaling, and a modular plugin system for easy extension. Authentication features, such as OAuth, JWT, LDAP, and more, are all available out-of-the-box and it has security features such as ACLs and CORS to provide high levels of protection.

If you move beyond a few basic services or want to provide your API to the public, it's well worth considering Kong as your starting point.

See also

- For more information on **Lua NGINX API variables**, refer to <https://github.com/openresty/lua-nginx-module#ngxvarvariable>
- For more information on **OpenResty FAQ**, refer to <https://openresty.org/en/faq.html>
- You can visit the official website of **Kong** at <https://getkong.org/>

NGINX Plus – The Commercial Offering

In this chapter, we will cover the following:

- Installing NGINX Plus
- Real-time server activity monitoring
- Dynamic config reloading
- Session persistence

Introduction

While the open source version of NGINX is the one most people are familiar with, Nginx Inc also produces a paid, commercial variant with a number of additional features aimed at enterprise and large tier deployments. With features such as detailed live monitoring, application health checking for load balancers, and dynamic configuration reloading, there are compelling reasons to consider the Plus version.

While some may find the US \$2,500 starting point a steep jump over the open source version, these additional features pit it against commercial systems at over ten times the price. Suffice to say, once you get to the point where these features become paramount to your business, the price is well worth it. You'll also be supporting the continual development of NGINX, which most organizations with the Plus version still run.

Installing NGINX Plus

Like its open source counterpart, NGINX Plus can be easily installed using the official repositories provided by NGINX.

Getting ready

If you have NGINX installed already, you'll need to uninstall it first to prevent conflict. As NGINX Plus is a paid product, you'll also require a license key to complete the installation as well. This can be purchased from the NGINX store or you can request a trial license so that you can evaluate the features before buying.

How to do it...

Official repositories exist for most major Linux installations, but we'll focus on just CentOS 7 and Ubuntu 16.04 LTS to cover the two most common scenarios.

mkdir -p /etc/ssl/nginx

nginx-repo.key

nginx-repo.crt

yum install -y ca-certificates

**yum-config-manager --add-repo **
https://cs.nginx.com/static/files/nginx-plus-7.repo

yum install -y nginx-plus

nginx -v

nginx version: nginx/1.11.10 (nginx-plus-r12-p3)

systemctl enable nginx **systemctl start**
nginx

The installation of NGINX Plus is now complete.

```
<strong>mkdir -p /etc/ssl/nginx</strong>
```

```
<strong>nginx-repo.key</strong>
```

```
<strong>nginx-repo.crt</strong>
```

```
<strong>wget http://nginx.org/keys/nginx_signing.key && apt-key  
add \ <br/> nginx_signing.key</strong>
```

```
<strong>apt install -y software-properties-common apt-transport-https  
\<br/> lsb-release ca-certificates</strong>
```

```
<strong>add-apt-repository -y \<br/> "deb https://plus-  
pkgs.nginx.com/ubuntu $(lsb_release -sc) main"</strong>
```

```
<strong>wget -q -O /etc/apt/apt.conf.d/90nginx \<br/>  
https://cs.nginx.com/static/files/90nginx</strong>
```

```
<strong>apt update</strong>
```

```
<strong>apt install nginx-plus</strong>
```

```
<strong>nginx -v</strong>
```

```
<strong>nginx version: nginx/1.11.10 (nginx-plus-r12-p3)</strong>
```

```
<strong>systemctl enable nginx</strong> <strong>systemctl start  
nginx</strong>
```

The installation of NGINX Plus is now complete.

See also

- The NGINX Plus product page: <https://www.nginx.com/products/>
- The NGINX Plus installation guide: <https://www.nginx.com/resources/admin-guide/installing-nginx-plus/>

Real-time server activity monitoring

We covered some of the basics of monitoring your server all the way back in [Chapter 1, *Let's Get Started*](#), which used the `ngxtop` utility to provide basic command line driven monitoring and statistics. Included in NGINX Plus is a powerful and comprehensive metrics system to show you real-time activity for your NGINX server.

As you'll see in the upcoming screenshots, this is quite a comprehensive system. Not only does it include a web interface so that you can see the stats, but this data is also available as a JSON feed so that it can be directly imported by external monitoring tools.

nginx version: nginx/1.11.10 (nginx-plus-r12-p3)

The important part to check is the *plus* in the naming for nginx-plus-r12-p3.

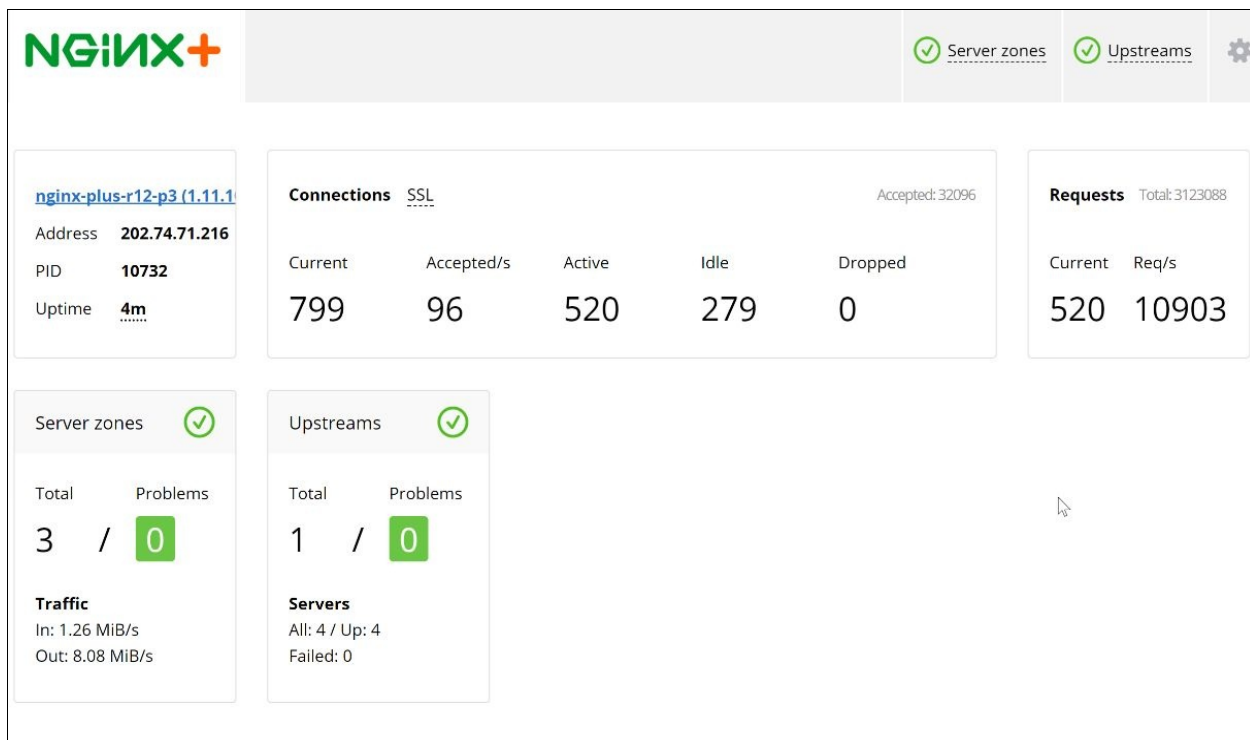
How to do it...

1. In order to use the status module, we need to include it as part of the `server` block directive. You can choose to run in on a different port from your main web server and also restrict via IP, which is what we'll do in our recipe. Here's our configuration:

```
server {  
    listen 8188;  
    status_zone status-page;  
  
    allow 192.168.0.0/24;  
    deny all;  
  
    root /usr/share/nginx/html;  
    location = /status.html { }  
    location = / {  
        return 301 /status.html;  
    }  
    location /status {  
        status;  
        status_format json;  
    }  
}
```

This can be placed in the `/etc/nginx/conf.d/` directory as a separate file (for example, `status.conf`), to keep it clean and simple.

2. If you open up a browser and go to `http://<serveripaddress>:8188/status.html`, you should see a page similar to the following screenshot:



This page is just an HTML page, but uses the JSON data to update the status (every second by default). We can see all of the vital server statistics, such as the connections, requests, uptime, and traffic.

If you have multiple server zones configured, you can view the stats individually to see the various statistics of your subsystems:

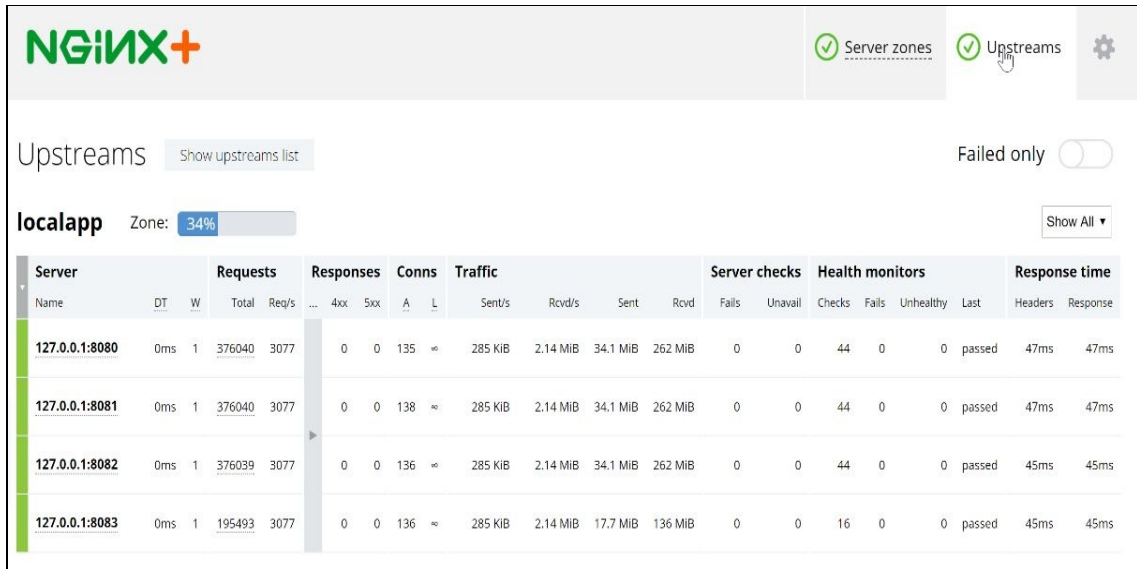
Server zones

Upstreams

Server zones

Zone	Requests			Responses						Traffic			
	Current	Total	Req/s	1xx	2xx	3xx	4xx	5xx	Total	Sent/s	Rcvd/s	Sent	Rcvd
main-app	505	1106967	11399	0	1106462	0	0	0	1106462	8.52 MiB	1.33 MiB	819 MiB	128 MiB
api-config	0	1	0	0	1	0	0	0	1	0	0	179 B	136 B
status-page	1	204	1	0	203	0	0	0	203	3.30 KiB	321 B	593 KiB	63.6 KiB

Likewise, we can see the stats associated with upstream servers if you're using NGINX Plus in a load balancer configuration:



These insights allow you to precisely see what's going on with all aspects of your server in real-time.

How it works...

We use the `listen` directive to tell NGINX Plus to listen on a different port for this `server` block, so that we can isolate it from the main connections. In this instance, we have it listening on port `8188`.

Next, we set `status_zone`, which needs to be defined within each `server` or `http` block directive in order to collect statistics. For this recipe, we simply have one zone within our statistics block directive called `status-page`. For other `server` block directives, you can either combine into one (for example, `backend-zone`) or track individually if you require unique statistics per directive.

In order to serve the static `status.html` file, we define the root path where the files are located. Then, we ensure any root calls (for example, without a trailing `/status.html`) are redirected.

Lastly, we set the `/status` location to serve the actual statistical data. We set this to be JSON format, which means it can be easily ingested into many other systems. It's what powers the HTML-based dashboard as well. This is required in order to display the statistics.

There's more...

The HTML dashboard is, of course, only one way to use the stats. Because we have direct access to the JSON data, we can pull this data from a third-party monitoring system. For instance, we can view what information is available by calling:

```
| curl http://202.74.71.216:8188/status
```

This will produce a JSON based output, similar to the following:

```
| {"version":8,"nginx_version":"1.11.10","nginx_build":"nginx-plus-r12-p3", "address":"200d
```

You can also reduce the output to the specific component you're after, for example, we can call `/status/connections` to retrieve just the statistics about the connections:

```
| {"accepted":945526,"dropped":0,"active":1,"idle":0}
```



Don't forget to adjust your accept/allow and/or authentication for third-party monitoring systems.

See also

The NGINX Plus status module documentation is available at: https://nginx.org/en/docs/http/ngx_http_status_module.html

Dynamic config reloading

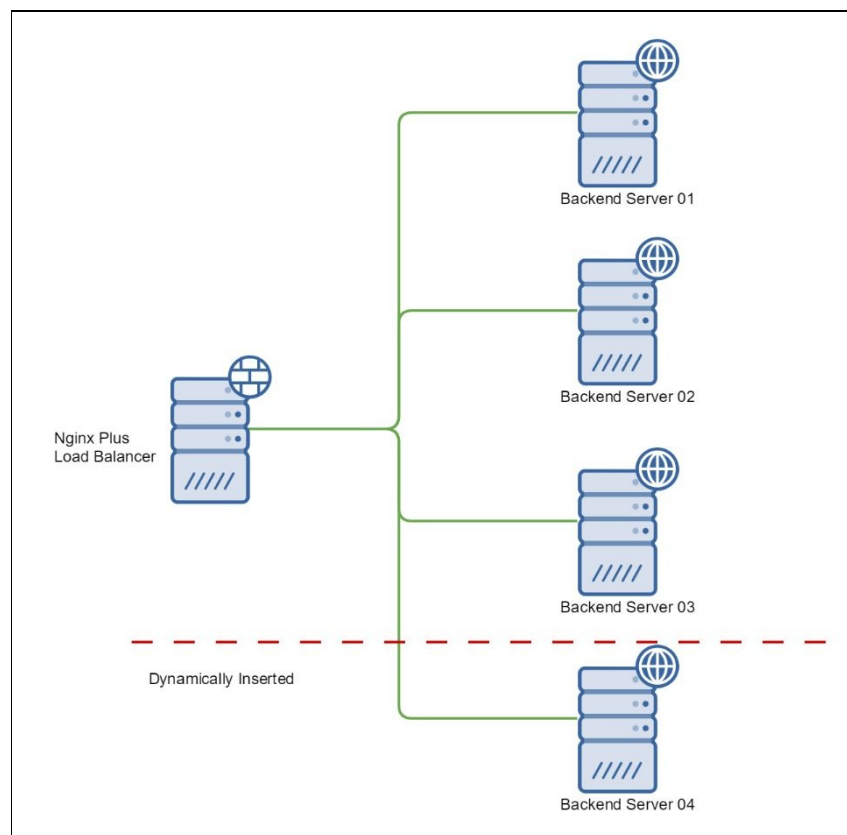
In large, high-scale systems where it's critical to minimize downtime, changes requiring a restart of services require careful planning. If you have a **Software as a Service (SaaS)** style platform, having a delay before deploying changes or new customer signups could be quite detrimental to your operations.

System downtime can also be required in load balancing situations, where you need to add and remove application backends on the fly. Thankfully, NGINX Plus allows for the configuration to be reloaded without having to restart the services. In this recipe, we'll go through how to update your configuration with a dynamic reload.

Getting ready

We're using a basic recipe based on our load balancing recipe back in [Chapter 8, Load Balancing](#). With the open source version of NGINX, adding or removing backend servers from an NGINX configuration required a full reload in order to use the new changes. While this only has a minimal impact on lightly loaded systems, this can be a significant problem when it comes to highly loaded systems. As it has to wait for all NGINX worker processes to finish before reloading, there's a period where the system won't be processing at the same capacity as normal.

Here's how our scenario will look:



```
upstream localapp {  
    zone backend 64k; server 127.0.0.1:8080; server 127.0.0.1:8081;  
    server 127.0.0.1:8082; }
```

```
server {  
  
    listen 80;  
  
    server_name dynamicload.nginxcookbook.com; access_log  
/var/log/nginx/dynamicload-access.log combined; location / {  
  
    proxy_pass http://localapp; }  
  
}
```

```
server {  
  
    listen 127.0.0.1:8189;  
  
    location /upstream_conf {  
  
    upstream_conf; }  
  
}
```

**<curl 'http://localhost:8189/upstream_conf?
upstream=localapp'>**

**<server 127.0.0.1:8080; # id=0> <server
127.0.0.1:8081; # id=1> <server 127.0.0.1:8082; #**

id=2

curl 'http://localhost:8189/upstream_conf?
add=&upstream=localapp&server=127.0.0.1:8083'

server 127.0.0.1:8083; # id=3

curl 'http://localhost:8189/upstream_conf?
remove=&upstream=localapp&id=1'

server 127.0.0.1:8080; # id=0 server
127.0.0.1:8082; # id=2 server 127.0.0.1:8083; #
id=3

How it works...

Similar to a standard NGINX load balancer configuration, we first define an `upstream` block directive. This defines a memory allocation to store the configuration so that it can be updated on the fly.

Next, we define our standard `server` block directive. This is as per a standard configuration, which simply proxies the connections to the upstream servers.

Lastly, we then define a separate block directive to handle the `upstream_conf` module. We use the separate `server` block directive so that we can bind it to a specific port on the localhost and prevent accidental exposure to the internet.

```
upstream localapp {  
    zone backend 64k;  
  
    state /var/lib/nginx/state/servers.state;  
}
```

We add the `state` directive, which is updated when there are any changes to the upstream configuration as well as being reread when NGINX restarts. All other server directives within that upstream must also be removed, as you can't combine static server configuration with a dynamic, stateful configuration. The state file itself is plain text (and uses the standard NGINX server directive format) but any direct manipulation of the file is strongly discouraged.

See also

The NGINX Plus `http_upstream_conf` module documentation is available at: https://nginx.org/en/docs/http/ngx_http_upstream_conf_module.html

Session persistence

If you have a scenario where you have a load balancer with multiple backend servers (as we covered back in [Chapter 8, Load Balancing](#)), there can be some tricky scenarios where session tracking would be difficult to implement. While using the hash-based algorithm can ensure requests from the same IP is routed to the same backend, this doesn't always ensure a balanced distribution of requests.

One of the key features for NGINX Plus is session persistence, where requests from the same client need to be sent to the same server for the life of that session. Also known as "sticky" sessions, this can be especially important when it comes to payment systems, where the sharing of information between backend servers can be quite restrictive.

Getting ready

We'll reuse our simple, round-robin load balancing scenario and incorporate the various session persistence options available in NGINX Plus.

How to do it...

There are three different methods to ensuring sticky sessions, each with their various pros and cons. Here's a quick summary:

- `cookie`: This method uses a cookie on the first request to store the tracking information in order to ensure subsequent requests are routed to the same backend server. As it means modifying the headers, it may not be compatible with all systems.
- `learn`: This is a stateful method, which relies on existing data within existing response headers to determine a unique identifier in which to track the requests. For example, most web frameworks have their own session ID, which can be leveraged for tracking. This means that no data modification is required.
- `route`: Lastly, we can route the request based on variables to explicitly choose the upstream server to use. While similar to the cookie method (route also uses a cookie to help track), the explicit choice of server can be beneficial when you have reasons to push to different clients to different servers. This could be used a "feature flag" method of routing clients to newer servers with differing features if they match the specific variables.

```
upstream stickyapp {
```

```
    server 127.0.0.1:8080; server 127.0.0.1:8081; server  
    127.0.0.1:8082; sticky cookie cookbook expires=1h; }
```

```
server {
```

```
    listen 80;
```

```
    server_name session.nginxcookbook.com; access_log  
    /var/log/nginx/sessiontest-access.log combined; location / {
```

```
        proxy_pass http://stickyapp; }
```

```
    }
```

```
<strong>http http://session.nginxcookbook.com/</strong>
```

```
<strong>HTTP/1.1 200 OK</strong> <strong>Connection: keep-  
alive</strong> <strong>Content-Length: 613</strong>
```

```
<strong>Content-Type: text/html; charset=utf-8</strong>
```

```
<strong>Date: Mon, 18 Aug 2017 15:39:37 GMT</strong>
```

```
<strong>Server: nginx/1.11.10</strong> <strong>Set-Cookie:  
cookbook=df2b80dc43705d28db9be6f29fe58da3; expires=Mon,<br/>  
18-Aug-17 16:39:37 GMT</strong>
```

```
upstream stickyapp {
```

```
    server 127.0.0.1:8080; server 127.0.0.1:8081; server  
127.0.0.1:8082; sticky learn create=$upstream_cookie_sessionid  
<br/> lookup=$cookie_sessionid zone=client_sessions:1m; }
```

```
server {
```

```
    listen 80; server_name session.nginxcookbook.com; access_log  
/var/log/nginx/sessiontest-access.log combined; location / {
```

```
    proxy_pass http://stickyapp; }
```

```
}
```

2. To test, you'll need to ensure that your application is sending a cookie set as sessionid.

```
map $cookie_route $route_cookie {
```

```
    ~(?P<route>\w+)$ $route; }
```

```
map $arg_route $route_uri {
```

```
    ~(?P<route>\w+)$ $route; }
```

```
upstream stickyapp {
```

```
    server 127.0.0.1:8080 route=server1; server 127.0.0.1:8081  
route=server2; server 127.0.0.1:8082 route=server3; sticky route  
$route_cookie $route_uri; }
```

```
server {
```

```
    listen 80;
```

```
    server_name session.nginxcookbook.com; access_log  
/var/log/nginx/sessiontest-access.log combined; location / {
```

```
        proxy_pass http://stickyapp; }
```

```
    status_zone sticky-app; }
```

```
<strong>curl --cookie 'route=server2'  
http://session.nginxcookbook.com/</strong>
```

Each request should be served from the upstream server tagged

as server2, which you can verify with the HTest utility or the logs of your backend server.

How it works...

For our cookie-based session tracking, we set a cookie named `cookbook` to use as the tracking for session persistence. This is also set with an expiry of one hour. It's also possible to explicitly set the domain and path as well if greater restrictions are required.

Our learn-based tracking has three variables set. The first, `create` is used to track from the `upsteam` server, which we look for the cookie set header (`Set-Cookie`) using a naming pattern `$upstream_cookie_<cookienname>`. For our recipe, `$upstream_cookie_sessionid` means we match the `sessionid`.

Next, we use the `lookup` variable to designate what to track from the client. This uses a similar tracking method to the `create` command. This recipe uses the `$cookie_sessionid` pattern, which means it will match the contents of a cookie named `sessionid`.

Lastly, as this is stateful, we need to allocate memory in which to store the lookup table. This is done via the `zone` variable. For this recipe we have named the zone `client_sessions` and allocated 1 megabyte of memory. This is sufficient to store around 8,000 sessions. By default, these sessions are only persistent for 10 minutes. So, if you have a higher number of users per 10 minutes or require a longer timeout, you may need to increase the memory allocated.

See also

The NGINX Plus `http_upstream_conf` module documentation is available at: https://nginx.org/en/docs/http/ngx_http_upstream_conf_module.html#sticky