packt

5TH EDITION

NGINX HTTP Server

Harness the power of NGINX with a series of detailed tutorials and real-life examples

GABRIEL OUIRAN | CLEMENT NEDELCU
MARTIN BJERRETOFT FJORDVALD

NGINX HTTP Server

Harness the power of NGINX with a series of detailed tutorials and real-life examples

Gabriel Ouiran

Clement Nedelcu

Martin Bjerretoft Fjordvald

NGINX HTTP Server

The author acknowledges the use of cutting-edge AI, such as ChatGPT, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Group Product Manager: Dhruv J. Kataria

Publishing Product Manager: Khushboo Samkaria

Book Project Manager: Srinidhi Ram

Senior Editor: Adrija Mitra

Technical Editor: Nithik Cheruvakodan

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Hemangini Bari

Production Designer: Shankar Kalbhor

DevRel Marketing Coordinator: Shruthi Shetty

Senior DevRel Marketing Executive: Marylou De Mello

First published: July 2010
Second edition: July 2013
Third edition: November 2015
Fourth edition: February 2018
Fifth edition: June 2024
Production reference: 1080524

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83546-987-3

www.packtpub.com

To the incredible circle of friends and mentors who've guided and helped me, to the community that has always inspired me, and to all those who hold a special place in my heart, filling my life with joy and purpose – you truly are my pillars of strength.

– Gabriel Ouiran

Contributors

About the authors

Gabriel Ouiran is a systems administrator at Eduka Software, Singapore, specializing in IT infrastructure with a strong focus on AI and cloud technologies. Gabriel is also a passionate tech enthusiast and a free and open source software contributor. He has over 10 years of experience working in IT for different companies. Gabriel now applies this expertise to enhance educational software solutions.

I'm grateful to Clément for the chance to write this fifth edition. Special thanks to Remy, Jean-Vincent, and Inès for their invaluable support and assistance in writing this book.

Clement Nedelcu was born in France and studied at UK, French, and Chinese universities. After teaching computer science, programming, and systems administration at several eastern Chinese universities, he worked as a technology consultant in France, specialized in the web and .NET software development as well as Linux server administration. Since 2005, he has also been administering a major network of websites in his spare time, which allowed him to discover NGINX. Clement now works as CTO. for a Singapore-based company developing management software for schools.

Martin Bjerretoft Fjordvald is a 35-year-old Danish entrepreneur who started his company straight out of high school. Backed by a popular website, he became a jack of all trades having to deal with the business, programming, and marketing side of his business. The popularity of his website grew and so did the performance requirements of his code and servers.

He got involved with the community project to document NGINX early on and has written several blog posts and wiki articles detailing how NGINX works.

About the reviewers

Rémy Beaufils has worked in the field of IT security for the last 10 years. He has performed security audit and consulting for both the private and public sector, from small local companies to international conglomerates. His current job was as security engineer for a French Insurtech start-up named Luko.

Rémy is also a contributor to several open source projects linked to automation and social media, and he is an active participant in several security-oriented communities.

I believe that no matter how small the participation, all of us can build a better digital future in the open source space, far from the conflicts of interest moving the private sector.

I'd also like to thank Ronnie, my cat, who helped me review and test all chapters of this book.

Jean-Vincent HAY has been a security auditor and DevSecOps consultant for more than eight years and currently works for Squad in France. His main areas of interest are application security, especially on mobile applications and their backends, and he mainly works for the industrial and banking sectors.

I hope this book helps people do a better job at giving me less work in raising awareness about OWASP and the whole security auditing industry. I'd like to thank Squad for giving me so many occasions to learn about good and bad NGINX configuration, and a community of pirate octopuses for their heads-up. Of course, this would have been much more difficult without the love of my life.

Table of Contents

2

Basic NGINX Configuration 23

Part 2: Dive into NGINX

3

Exploring the HTTP Configuration 47

4

Exploring Module Configuration in NGINX 87

5

PHP and Python with NGINX 139

6

NGINX as a Reverse Proxy 159

Part 3: NGINX in Action

7

Introduction to Load Balancing and Optimization 177

8

NGINX within a Cloud Infrastructure 189

9

Fully Deploy, Manage, and Auto-Update NGINX with Ansible 199

10

11

Preface

NGINX, known for its lightweight design, is a powerhouse HTTP server designed for handling high-traffic websites, with network scalability as its primary objective. In our increasingly connected world, optimizing your web apps has become more critical than ever. Whether you are a beginner or an experienced administrator, this NGINX book will guide you through the complete process of setting up this lightweight HTTP server, from a quick and basic configuration to a more detailed configuration tailored to your needs. This new edition focuses on the latest version 1.25.2, which introduces new features such as HTTP/3 and QUIC.

Packed with a multitude of real-world examples, this book will help you to secure your infrastructure with automatic TLS certificates, expertly place NGINX in front of your existing applications, and do much more. You'll also cover orchestration, Docker, bandwidth management, OpenResty, and NGINX Plus commercial features to enhance and optimize your infrastructure or design a brand-new architecture.

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

Who this book is for

This edition of our NGINX book is meticulously designed for beginner DevOps engineers, system administrators, and web developers who want to improve their understanding of web server management, especially in the areas of performance optimization and cloud infrastructure. Whether you're just beginning your journey into web technologies or looking to consolidate your base, this book offers practical insights and hands-on experience with NGINX, the heart of modern web applications.

The primary personas who will find this content invaluable are beginner DevOps engineers, system administrators, and curious web developers. DevOps newcomers will learn NGINX's essentials, from setup to cloud optimization, while system administrators will gain strategies for effective deployment and troubleshooting, and developers will discover how to enhance their projects with improved performance, security, and scalability.

What this book covers

Chapter 1, *Downloading and Installing NGINX*, is an introduction to acquiring and configuring NGINX on your system. It covers the essentials, from the initial download to ensuring NGINX persists across reboots by installing it as a system service. Key areas include preparing your system, configuring NGINX, managing dynamically loaded modules, and integrating NGINX as a reliable system service. The chapter closes with an overview of NGINX Plus.

Chapter 2, *Basic NGINX Configuration*, explores the essentials of NGINX configuration, from file syntax and structure to testing changes. Highlights include configuration syntax, base directives, and server testing strategies.

Chapter 3, *Exploring the HTTP Configuration*, focuses on the intricacies of the NGINX HTTP module, providing a comprehensive guide to directives and an in-depth exploration of the Location block. From the fundamentals of the HTTP Core module, through in-depth discussions of directives and variables, to mastery of the Location block, this chapter provides readers with the know-how to efficiently configure virtual hosts and understand the principles underlying NGINX's configuration structure.

Chapter 4, *Exploring Module Configuration in NGINX*, delivers an in-depth guide to NGINX's Rewrite module and a comprehensive overview of the open source edition's first-party modules. This chapter focuses on mastering "pretty URLs" with the Rewrite module, and exploring NGINX's vast selection of add-on modules.

Chapter 5, *PHP and Python with NGINX*, emphasizes configuring NGINX to work with dynamic PHP and Python scripts via FastCGI. Starting with a deep dive into the principles of FastCGI, it then walks through the process of configuring NGINX for efficient communication with PHP and Python applications, offering insights into using the FastCGI module for optimal server-side script integration.

Chapter 6, *NGINX as a Reverse Proxy*, discusses NGINX's reverse proxy capabilities and dives into the details of the NGINX proxy module. Readers will learn how to navigate the proxy module, appreciate NGINX's strengths in managing modern web applications, and set up a reverse proxy configuration efficiently.

Chapter 7, *Introduction to Load Balancing and Optimization*, focuses on deploying NGINX to manage high-traffic websites on multiple servers. This chapter looks at NGINX's advanced load-balancing capabilities and the latest I/O optimization features, including the use of thread pools. It offers a comprehensive guide to setting up a balanced server architecture and improving I/O performance, crucial for the scalability and efficiency of large-scale web applications.

Chapter 8, *NGINX within a Cloud Infrastructure*, highlights the integration of NGINX and Docker in cloud setups, from Docker fundamentals to deploying NGINX with Docker Compose. Learn to optimize containerized applications and caching with NGINX as a central gateway.

Chapter 9, *Fully Deploy, Manage, and Auto-Update NGINX with Ansible*, explores efficient NGINX deployment across multiple servers using Ansible. Covering mass deployment strategies, the basics of configuration management, and crafting Ansible roles for NGINX, this chapter will help with large-scale server management. It concludes with automating NGINX updates for effortless maintenance to deliver a robust and secure NGINX fleet.

Chapter 10, *Case Studies*, dives into the deployment of secure websites and applications with detailed case studies. Master SSL with acme.sh, adopt HTTP/2, configure WordPress and integrate NextCloud with Docker. Gain skills in deploying secure sites, configuring NGINX for various scenarios and using NGINX to enhance Docker environments.

Chapter 11, *Troubleshooting*, navigates through NGINX debugging with essential tips and tools for identifying issues. Covering general troubleshooting advice, log parsing, and common challenges such as install errors and 403 Forbidden pages, this chapter lays out a clear path to resolving configuration mistakes and understanding block priorities. Learn to adjust configurations for effective debugging and recognize frequent NGINX configuration errors.

To get the most out of this book

Operating system requirements	**Software/hardware covered in the book**
Linux (preferably Debian, Ubuntu, Fedora, or RHEL)	NGINX
	Ansible
	Docker

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/NGINX-HTTP-Server_Fifth-edition`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/OS/OSRELEASE/$basearch/
gpgcheck=0
enabled=1
```

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

```
apt update
apt install nginx
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Select **System info** from the **Administration** panel."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *NGINX HTTP Server*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835469873`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Begin with NGINX

In this part, you will be introduced to the foundations of NGINX. This section provides the basis for understanding NGINX and its capabilities by exploring the installation and the baseline configuration of the web server. You will get practical information on the efficient configuration for NGINX, which will prepare you for the advanced configuration and optimization techniques covered in the next parts.

This part includes the following chapters:

- *Chapter 1*, *Downloading and Installing NGINX*
- *Chapter 2*, *Basic NGINX Configuration*

1
Downloading and Installing NGINX

NGINX (pronounced *engine-x*) has become the leader among web servers ever since it came out 20 years ago. Back in 2004, its main goal was to outperform Apache, and today, NGINX is outperforming every web server when it comes to high-traffic websites or security. Throughout this book, we will discover and learn how to use NGINX, step by step. We will cover many subjects to have a configuration tailored to everyone's needs.

In this first chapter, we will proceed with the necessary steps toward establishing a functional setup of NGINX. This moment is crucial for the smooth functioning of your web server—there are some required libraries and tools for installing the web server, some parameters that you will have to decide upon when compiling the binaries, and there may also be some configuration changes to perform on your system.

By the end of this chapter, you will have installed NGINX either through a public repository or by compiling a custom version embedding all the extra modules you might need.

This chapter covers the following:

- Installing NGINX via package managers
- Downloading and installing the prerequisites for compiling NGINX binaries
- Downloading a suitable version of the NGINX source code
- Configuring NGINX compile-time options
- Controlling the application with a `unit service` file
- Configuring the system to launch NGINX automatically on startup
- A quick overview of the possibilities offered by NGINX Plus

…

Installing NGINX via package managers

The quickest, and easiest, way to install NGINX is to simply use your OS-provided version. Most of the time, these are kept fairly updated; however, for some Linux distributions focusing on stability, you may only have older versions of NGINX available. Sometimes, your Linux distribution may provide multiple versions of NGINX with different compile flags.

In general, before embarking on a more complex journey, we should check whether we can use the easy solution. For Red Hat Linux-based operating systems, we need to enable the EPEL repo first and then do the same:

```
yum install epel-release
yum search nginx
yum info PACKAGE_NAME
yum install PACKAGE_NAME
```

For a Debian-based operating system, we first find the NGINX compiles available and then get the information for the one we want:

```
apt-cache search nginx
apt-cache show PACKAGE_NAME
apt install PACKAGE_NAME
```

If the version provided is current enough, then you're ready to configure NGINX in the next chapter.

If the version provided by your distribution is too old, then NGINX provides packages for **RHEL/CentOS distributions** as well as **Debian/Ubuntu distributions**. We encourage you to visit the official NGINX website to make sure the version given by your distribution isn't outdated.

NGINX-provided packages

To set up a `yum` repository for RHEL/CentOS, create a file named `/etc/yum.repos.d/nginx.repo` with the following contents:

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/OS/OSRELEASE/$basearch/
gpgcheck=0
enabled=1
```

Replace `OS` with `rhel` or `centos`, depending on the distribution used, and `OSRELEASE` with `8` or `9`, for versions 8.x or 9.x, respectively. Afterward, NGINX can now be installed with `yum`:

```
yum install nginx
```

For Debian-based distributions, we need to first use their signing key to authenticate the package signatures. Download the following file first from `http://nginx.org/keys/nginx_signing.key`.

Then, run the following command:

```
sudo apt-key add nginx_signing.key
```

With the key added, we can now add the NGINX repository to `sources.list` found in `/etc/apt/sources.list`. For Debian, we add the following lines:

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

Here, `codename` is either `trixie` or `bookworm`, depending on your version of Debian. For Ubuntu, we use the following dependencies:

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

Here, `codename` is either `noble`, `focal`, or `bionic`, depending on your version of Ubuntu. Finally, we can install NGINX with the `apt` command:

```
apt update
apt install nginx
```

Now that we have learned how to install NGINX from repositories, let's have a look at how we can compile it from the source and benefit from having custom modules that are not provided with the default NGINX.

Compiling NGINX from source

There are situations where compiling NGINX from source is preferable. It gives us the most flexibility regarding modules, so we can customize better for our intended usage. For example, we could compile a very lean version for embedded hardware.

Additionally, we can make sure we use the latest version of NGINX and have all new features available to us. Keep in mind, though, that when installing software from source, you are responsible for keeping it updated. NGINX, just like every other piece of software, sometimes finds security issues that it needs to address. An OS package is much easier to update than a source installation but, so long as you're aware of the need to maintain it yourself, there is absolutely no problem.

Depending on the optional modules that you select at compile time, you will perhaps need different prerequisites. We will guide you through the process of installing the most common ones, such as **GCC**, **PCRE**, **zlib**, and **OpenSSL**.

Installing GNU Compiler Collection

NGINX is a program written in C, so you will first need to install a compiler tool such as the **GNU Compiler Collection** (**GCC**) on your system. GCC may already be present on your system, but if that is not the case, you will have to install it before going any further.

> **Note**
>
> GCC is a collection of free open source compilers for various languages – C, C++, Java, Ada, Fortran, and so on. It is the most commonly used compiler suite in the Linux world, and Windows versions are also available. A vast number of processors are supported, such as x86, AMD64, PowerPC, ARM, MIPS, and more.

Follow these steps to install GCC:

1. First, make sure it isn't already installed on your system:

   ```
   [user@server ~]$ gcc
   ```

   If you get the following output, it means that GCC is correctly installed on your system and you can skip to the next section:

   ```
   gcc: no input files
   ```

   If you receive the following message, you will have to proceed with the installation of the compiler:

   ```
   ~bash: gcc: command not found
   ```

2. GCC can be installed using the default repositories of your package manager. Depending on your distribution, the package manager will be `vary-yum` for a Red Hat Linux-based distribution, `apt` for Debian and Ubuntu, `yast` for SUSE Linux, and so on. Here is the typical way to proceed with the download and installation of the GCC package:

   ```
   [root@server ~]# yum groupinstall "Development Tools"
   ```

3. If you use `apt`, execute the following command:

   ```
   [root@server ~]# apt-get install build-essentials
   ```

   If you use another package manager with a different syntax, you will probably find the documentation with the `man` utility. Either way, your package manager should be able to download and install GCC correctly, after having resolved dependencies automatically.

Note that the `apt` and `yum` commands will not only install GCC; they will also proceed to download and install all common requirements for building applications from source, such as code headers and other compilation tools.

The PCRE library

The **Perl Compatible Regular Expressions** (**PCRE**) library is required for compiling NGINX. The rewrite and HTTP core modules of NGINX use PCRE for the syntax of their regular expressions, as we will discover in later chapters. You will need to install two packages—`pcre` and `pcre-devel`. The first one provides the compiled version of the library, whereas the second one provides development headers and sources for compiling projects, which are required in our case.

Here are some example commands that you can run in order to install both packages.

Using `yum`, execute the following command:

```
[root@server ~]# yum install pcre pcre-devel
```

Or you can install all PCRE-related packages using the following command:

```
[root@server ~]# yum install pcre*
```

If you use `apt`, use the following command:

```
[root@server ~]# apt install libpcre3 libpcre3-dev
```

If these packages are already installed on your system, you will receive a message saying something like `nothing to do`; in other words, the package manager did not install or update any component:

```
root@server:~# apt install libpcre3 libpcre3-dev
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
libpcre3 is already the newest version (2:8.39-13ubuntu0.22.04.1).
libpcre3-dev is already the newest version (2:8.39-13ubuntu0.22.04.1).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
root@server:~# _
```

Figure 1.1: APT showing the PCRE library is already installed.

The preceding output signifies that both the components, `libpcre3` and `libpcre3-dev`, are already present in the system.

The zlib library

The `zlib` library provides developers with compression algorithms. It is required for the use of `.gzip` compression in various modules of NGINX. Again, you can use your package manager to install this component as it is part of the default repositories. Similar to PCRE, you will need the `zlib` library and its associated `zlib-dev` component as well.

Using `yum`, execute the following command:

```
[root@server ~]# yum install zlib zlib-devel
```

Using `apt`, execute the following command:

```
[root@server ~]# apt install zlib1g zlib1g-dev
```

These packages install quickly and have no known dependency issues.

OpenSSL

The OpenSSL project is a collaborative effort to develop a robust, commercial-grade, full-featured, and open source toolkit implementing the **Secure Sockets Layer** (**SSL**) v2/v3 and **Transport Layer Security** (**TLS**) v1 protocols as well as a full-strength general-purpose cryptography library. The project is managed by a worldwide community of volunteers who use the internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation. For more information, visit `https://www.openssl.org`.

The OpenSSL library will be used by NGINX to serve secure web pages. We thus need to install the library and its development package. The process remains the same here – you install `openssl` and `openssl-devel`:

```
[root@server ~]# yum install openssl openssl-devel
```

Using `apt`, execute the following command:

```
[root@server ~]# apt install openssl libssl-dev
```

> **Important note**
>
> Please be aware of the laws and regulations in your own country. Some countries do not allow the use of strong cryptography. The author, publisher, and developers of the OpenSSL and NGINX projects will not be held liable for any violations or law infringements on your part.

Now that you have installed all of the prerequisites, you are ready to download and compile the NGINX source code.

Downloading and compiling the NGINX source code

This approach to the download process will lead us to discover the various resources at the disposal of server administrators, websites, communities, and wikis all relating to NGINX. We will also quickly discuss the different version branches available to you, and eventually, select the most appropriate one for your setup.

Websites and resources

Although NGINX is a relatively new and growing project, there are already a good number of resources available on the **World Wide Web** (**WWW**) and an active community of administrators and developers.

The official website, which is `https://nginx.org/`, currently serves as an official documentation reference and provides links from which to download the latest version of the application source code and binaries. A wiki is also available at `https://www.nginx.com/resources/wiki/` and offers a wide selection of additional resources such as installation guides for various operating systems, tutorials related to the different modules of NGINX, and more.

There are several ways to get help if you should need it. If you have a specific question, try posting on the NGINX forum at `https://forum.nginx.org/`. An active community of users will answer your questions in no time. Additionally, the NGINX mailing list, which is relayed on the NGINX forum, will also prove to be an excellent resource for any question you may have. If you need direct assistance, there is always a group of regulars helping each other out on the IRC channel, `#Nginx`, on `irc.libera.chat`.

Another interesting source of information is the blogosphere. A simple query on your favorite search engine should return a good number of blog articles documenting NGINX, its configuration, and modules:

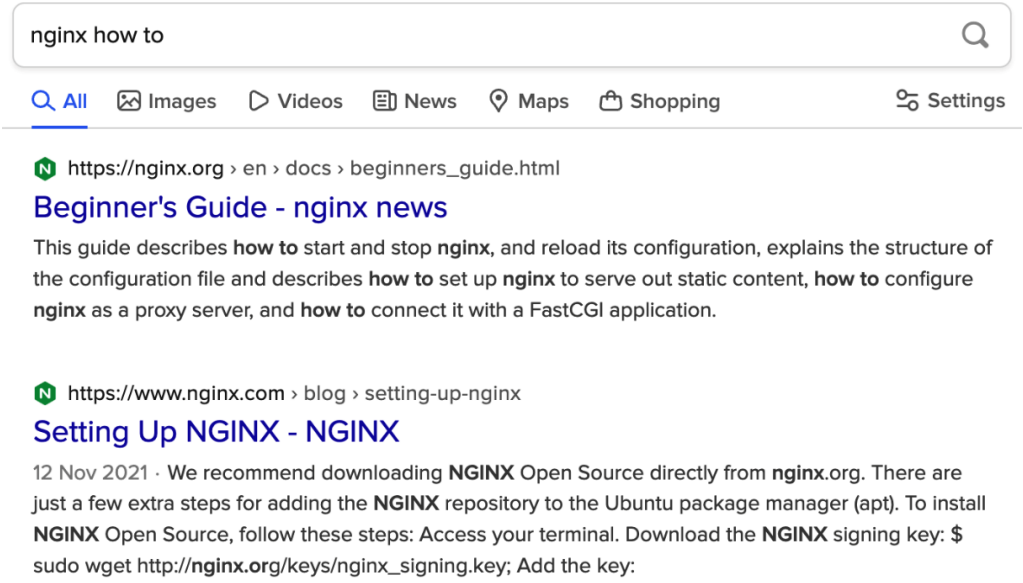


Figure 1.2: Websites and blogs documenting Nginx

It's now time to head over to the official website and get started with downloading the source code for compiling and installing NGINX. Before you do so, let us have a quick summary of the available versions and the features that come with them.

Version branches

Igor Sysoev, a talented Russian developer and server administrator, initiated this open source project back in 2002. Between the first release in 2004 and the current version, the market share of NGINX has been growing steadily. It now serves nearly 26.23% of websites on the internet, according to an April 2023 survey on `https://www.netcraft.com/`. The features are numerous and render the application both powerful and flexible at the same time.

There are currently three version branches of the project:

- **Stable version**: This version is usually recommended, as it is approved by both developers and users, but is usually a little behind the mainline version.
- **Mainline version**: This is the latest version available for download and comes with the newest developments and bug fixes. It was formerly known as the **development version**. Although it is generally solid enough to be installed on production servers, there is a small chance that you will run into the occasional bug. As such, if you favor stability over novelty, going for the stable version is recommended.
- **Legacy version**: If, for some reason, you are interested in looking at the older versions, you will find several of them.

A recurrent question regarding mainline versions is "*Are they stable enough to be used on production servers?*" Cliff Wells, the original founder and maintainer of the NGINX wiki at `https://www.nginx.com/resources/wiki/`, believes so – "*I generally use and recommend the latest development version. It's only bit me once!*" Early adopters rarely report critical problems. It is up to you to select the version you will be using on your server, knowing that the instructions given in this book should be valid regardless of the release as the NGINX developers have decided to maintain overall backward compatibility in new versions. You can find more information on version changes, new additions, and bug fixes on the dedicated change log page on the official website.

Features

As of the mainline version 1.25.2, NGINX offers an impressive variety of features, which, contrary to what the title of this book indicates, are not all related to serving HTTP content. Here is a list of the main features of the web branch, quoted from the official website (`https://nginx.org/`):

- Serving static and index files, auto indexing; open file descriptor cache; accelerated reverse proxying with caching; load balancing and fault tolerance.
- Accelerated support with caching of FastCGI, uWSGI, SCGI, and memcached servers; load balancing and fault tolerance; modular architecture. Filters include gzipping, byte ranges, chunked responses, XSLT, SSI, and image transformation filter. Multiple SSI inclusions within a single page can be processed in parallel if they are handled by proxies or FastCGI/uWSGI/SCGI servers.
- SSL and TLS SNI support.

NGINX can also be used as a mail proxy server. Although this aspect will not be closely documented in the book, the following will provide you with some insight into it:

- User redirection to IMAP/POP3 backend using an external HTTP authentication server
- User authentication using an external HTTP authentication server and connection redirection to an internal SMTP backend
- Authentication methods:
  - **POP3**: `USER/PASS, APOP, AUTH LOGIN/PLAIN/CRAM-MD5`
  - **IMAP**: `LOGIN, AUTH LOGIN/PLAIN/CRAM-MD5`
  - **SMTP**: `AUTH LOGIN/PLAIN/CRAM-MD5`
  - **SSL support**
  - **STARTTLS and STLS support**

NGINX is compatible with most computer architectures and operating systems—Windows, Linux, macOS, FreeBSD, and Solaris. The application runs fine on 32- and 64-bit architectures.

Downloading and extracting

Once you have made your choice as to which version you will be using, head over to `https://nginx.org/` and find the URL of the file you wish to download. Position yourself in your `home` directory, which will contain the source code to be compiled, and download the file using `wget`:

```
[user@server ~]$ mkdir src && cd src
[user@server src]$ wget https://nginx.org/download/nginx-1.25.2.tar.gz
```

We will be using version `1.25.2`, the latest stable version as of September 2023. Once downloaded, extract the archive contents in the current folder:

```
[user@server src]$ tar zxf nginx-1.25.2.tar.gz
```

You have successfully downloaded and extracted NGINX. Now, the next step will be to configure the compilation process in order to obtain a binary that perfectly fits your operating system.

Exploring the options for configuring the compilation

There are usually three steps when building an application from source—*configuration*, *compilation*, and *installation*. The configuration step allows you to select a number of options that will not be editable after the program is built, as it has a direct impact on the project binaries. Consequently, it is a very important stage that you need to follow carefully if you want to avoid surprises later, such as the lack of a specific module or files being located in a random folder.

The process consists of appending certain switches to the `configure` script that comes with the source code. We will discover the three types of switches that you can activate, but let us first study the easiest way to proceed.

The easy way

If, for some reason, you do not want to bother with the configuration step, such as for testing purposes or simply because you will be recompiling the application in the future, you may simply use the `configure` command with no switches. Execute the following three commands to build and install a working version of NGINX:

```
[user@server nginx-1.25.2]# ./configure
```

Running this command should initiate a long procedure of verifications to ensure that your system contains all of the necessary components. If the configuration process fails, please make sure you check the prerequisites section again, as it is the most common cause of errors. For information about why the command failed, you may also refer to the `objs/autoconf.err` file, which provides a more detailed report. The `make` command will compile the application:

```
[user@server nginx-1.25.2]# make
```

This step should not cause any errors as long as the configuration went fine:

```
[root@server nginx-1.25.2]# make install
```

This last step will copy the compiled files as well as other resources to the installation directory, by default `/usr/local/nginx`. You may need to be logged in as `root` to perform this operation depending on permissions granted to the `/usr/local` directory.

Again, if you build the application without configuring it, you take the risk of missing out on a lot of features, such as the optional modules and others that we are about to discover.

Path options

When running the `configure` command, you are offered the chance to enable some switches that let you specify the directory or file paths for a variety of elements. Please note that the options offered by the configuration switches may change according to the version you downloaded. The following options listed are valid with the stable version, as of release 1.25.2. If you use another version, run the `./configure --help` command to list the available switches for your setup.

Using a switch typically consists of appending some text to the command line. Here is an example using the `--conf-path` switch:

```
[root@server nginx]# ./configure --conf-path=/etc/nginx/nginx.conf
```

Most of the time, the default configuration switches do not need customization. However, it is highly recommended to have a look at the configuration switches in the official documentation page, **Building NGINX from Sources**, in order to configure the right paths for your logs, modules such as `https`, and libraries such as `geoip`, `gzip`, `zlib`, or `pcre`. Do note that, in this book, we will compile NGINX with the standard switches from **Building NGINX from Sources** and we will be aligned with the compiled binaries used by the popular Linux distributions.

> **Note**
>
> Be aware that these configurations do not include additional third-party modules. Please refer to *Chapter 5*, for more information about installing add-ons.

Build configuration issues

In some cases, the `configure` command may fail – after a long list of checks, you may receive a few error messages on your terminal. In most (if not all) cases, these errors are related to missing prerequisites or unspecified paths.

In such cases, proceed with the following verifications carefully to make sure you have all it takes to compile the application, and optionally consult the `objs/autoconf.err` file for more details about the compilation problem. This file is generated during the `configure` process and will tell you exactly which part of the process failed.

Make sure you installed the prerequisites

There are basically four main prerequisites: GCC, PCRE, `zlib`, and OpenSSL. The last three are libraries that must be installed in two packages: the library itself and its development sources. Make sure you have installed both for each of them. Please refer to the prerequisites at the beginning of this chapter. Note that other prerequisites, such as `LibXML2` or `LibXSLT`, might be required to enable extra modules (for example, in the case of the HTTP XSLT module).

If you are positive that all of the prerequisites were installed correctly, perhaps the issue comes from the fact that the `configure` script is unable to locate the prerequisite files. In that case, make sure that you include the configuration switches related to file paths, as described earlier.

For example, the following switch allows you to specify the location of the OpenSSL library files:

```
./configure [...] --with-openssl=/usr/lib64
```

The OpenSSL library file will be looked for in the specified folder.

Directories exist and are writable

Always remember to check the obvious; everyone makes even the simplest of mistakes sooner or later. Make sure the directory you placed the NGINX files in has *read* and *write* permissions for the user running the configuration and compilation scripts. Also ensure that all paths specified in the `configure` script switches are existing, valid paths.

Compiling and installing

The configuration process is of the utmost importance—it generates a makefile for the application depending on the selected switches and performs a long list of requirement checks on your system. Once the `configure` script is successfully executed, you can proceed with compiling NGINX.

Compiling the project equates to executing the `make` command in the project source directory:

```
[user@server nginx-1.25.2]$ make
```

A successful build should result in a final message appearing: `make[1]: leaving directory` followed by the project source path.

Again, problems might occur at compile time. Most of these problems can originate from missing prerequisites or invalid paths specified. If this occurs, run the `configure` command again and triple-check the switches and all of the prerequisite options. It may also be that you downloaded a too-recent version of the prerequisites that might not be backward-compatible. In such cases, the best option is to visit the official website of the missing component and download an older version.

If the compilation process was successful, you are ready for the next step: installing the application. The following command must be executed with `root` privileges:

```
[root@server nginx-1.25.2]# make install
```

The `make install` command executes the `install` section of the makefile. In other words, it performs a few simple operations, such as copying binaries and configuration files to the specified `install` folder. It also creates directories for storing log and HTML files, if these do not already exist. The `make install` step is not generally a source of problems unless your system encounters an exceptional error, such as a lack of storage space or memory.

> **Note**
>
> You might require `root` privileges for installing the application in the `/usr/local/` folder, depending on the folder permissions.

NGINX is now ready as it has been compiled successfully. In the next section, we will turn NGINX into a daemon running in the background.

Controlling the NGINX service

At this stage, you should have successfully built and installed NGINX. The default location for the output files is `/usr/local/nginx`, so we will be basing future examples using this path to start, stop, run at boot, and keep an eye on the NGINX status using a daemon.

Daemons and services

The next step is obviously to execute NGINX. However, before doing so, it's important to understand the nature of this application. There are two types of computer applications—those that require immediate user input, thus running in the foreground, and those that do not, thus running in the background. NGINX is of the latter type, often referred to as daemon. Daemon names usually come with a trailing `d` and a couple of examples can be mentioned here—`httpd` (the HTTP server daemon) is the name given to Apache under several Linux distributions, and `named` is the name server daemon. `cron` is the task scheduler—although, as you will notice, it is not the case for NGINX. When started from the command line, a daemon immediately returns the prompt window, and in most cases, does not even bother outputting data to the terminal.

Consequently, when starting NGINX, you will not see any text appear on the screen and the prompt will return immediately. While this might seem startling, it is, on the contrary, a good sign. It means the daemon was started correctly and the configuration did not contain any errors.

Users and groups

It is of the utmost importance to understand the process architecture of NGINX, particularly the users and groups its various processes run under. A very common source of trouble when setting up NGINX is invalid file access permissions—due to a user or group misconfiguration, you often end up getting `403 Forbidden` HTTP errors because NGINX cannot access the requested files.

There are two levels of processes with possibly different permission sets:

- **NGINX master process**: This should be started as `root`. In most Unix-like systems, processes started with the `root` account are allowed to open TCP sockets on any port, whereas other users can only open listening sockets on a port above `1024`. If you do not start NGINX as `root`, standard ports such as `80` or `443` will not be accessible.

> **Note**
>
> The user directive that allows you to specify a different user and group for the worker processes will not be taken into consideration for the master process.

- **NGINX worker processes**: These are automatically spawned by the master process under the account you specified in the configuration file with the user directive (detailed in *Chapter 2*, *Basic Nginx Configuration*). The configuration setting takes precedence over the configuration switch you may have specified at compile time. If you did not specify any of those, the worker processes will be started as `user nobody`, and the group will be nobody (or `nogroup`, depending on your OS).

NGINX command-line switches

The NGINX binary accepts command-line arguments for performing various operations, among which is controlling background processes. To get a full list of commands, you may invoke the **Help** screen using the following commands:

```
[user@server ~]$ cd /usr/local/nginx/sbin
[user@server sbin]$ ./nginx -h
```

The next few sections will describe the purpose of these switches. Some allow you to control the daemon, and some let you perform various operations on the application configuration.

Starting and stopping the daemon

You can start NGINX by running the NGINX binary without any switches. If the daemon is already running, a message will show up indicating that a socket is already listening on the specified port:

```
[emerg]: bind() to 0.0.0.0:80 failed (98: Address already in use)
[...] [emerg]: still could not bind().
```

Beyond this point, you may control the daemon by stopping it, restarting it, or simply reloading its configuration. Controlling is done by sending signals to the process using the `nginx -s` command:

Command	Description
`nginx -s stop`	Stops the daemon immediately (using the `TERM` signal)
`nginx -s quit`	Stops the daemon gracefully (using the `QUIT` signal)
`nginx -s reopen`	Reopens log files
`nginx -s reload`	Reloads the configuration

Table 1.1: A table to remember how to control the nginx daemon

When starting the daemon, stopping it, or performing any of the preceding operations, the configuration file is first parsed and verified. If the configuration is invalid, whatever command you have submitted will *fail*, even when trying to stop the daemon. In other words, in some cases, you will not be able to even stop NGINX if the configuration file is invalid.

An alternate way to terminate the process, in desperate cases only, is to use the `kill` or `killall` commands with `root` privileges:

```
[root@server ~]# killall nginx
```

Testing the configuration

As you can imagine, this tiny bit of detail might become an important issue if you constantly tweak your configuration. The slightest mistake in any of the configuration files can result in a loss of control over the service—you are then unable to stop it via regular `init` control commands, and obviously, it will refuse to start again.

In consequence, the following command will be useful to you on many occasions. It allows you to check the syntax, validity, and integrity of your configuration:

```
[user@server ~]$ /usr/local/nginx/sbin/nginx -t
```

The `-t` switch stands for **test configuration**. NGINX will parse the configuration anew and let you know whether it is valid or not. A valid configuration file does not necessarily mean NGINX will start though as there might be additional problems such as socket issues, invalid paths, or incorrect access permissions.

Obviously, manipulating your configuration files while your server is in production is a dangerous thing to do and should be avoided when possible. The best practice, in this case, is to place your new configuration into a separate temporary file and run the test on that file. NGINX makes it possible by offering the `-c` switch:

```
[user@server sbin]$ ./nginx -t -c /home/user/test.conf
```

This command will parse `/home/user/test.conf` and make sure it is a valid NGINX configuration file. When you are done, after making sure that your new file is valid, proceed to replace your current configuration file and reload the server configuration:

```
[user@server sbin]$ cp -i /home/user/test.conf usr/local/nginx/conf/
nginx.conf
cp: erase 'nginx.conf' ? yes
[user@server sbin]$ ./nginx -s reload
```

Other switches

Another switch that might come in handy in many situations is `-V`. Not only does it tell you the current NGINX build version but, more importantly, it also reminds you about the arguments that you used during the configuration step – in other words, the command switches that you passed to the `configure` script before compilation:

```
[user@server sbin]$ ./nginx -V
nginx version: nginx/1.25.2 (Ubuntu)
built by gcc 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
TLS SNI support enabled
configure arguments: --with-http_ssl_module
```

In this case, NGINX was configured with the `--with-http_ssl_module` switch only.

Why is this so important? Well, if you ever try to use a module that was not included with the `configure` script during the precompilation process, the directive enabling the module will result in a configuration error. Your first reaction will be to wonder where the syntax error comes from. Your second reaction will be to wonder if you even built the module in the first place! Running `nginx -V` will answer this question.

Additionally, the `-g` option lets you specify additional configuration directives, in case they were not included in the configuration file:

```
[user@server sbin]$ ./nginx -g "timer_resolution 200ms";
```

In the next section, we will configure the daemon through the system to have NGINX integrated and running automatically on your Linux distribution.

Adding NGINX as a system service

In this section, we will create a script that will transform the NGINX daemon into an actual system service. This will result in mainly two outcomes—the daemon will be controllable using standard commands and, more importantly, it will automatically be launched on system startup and stopped on system shutdown.

systemd unit file

Most Linux-based operating systems to date use a systemd-style *service file*. Debian, Ubuntu, RHEL, and centOS all use systemd nowadays; therefore, this service file should work on any popular linux distribution. There is other init software, such as System V and OpenRC; however, we will stick with the more popular and most supported init.

In this example, we will be using the NGINX service file provided by the official NGINX website:

1. Create a file by using the following command:

```
nano /etc/systemd/system/nginx.service
```

```
#/lib/systemd/system/nginx.service
[Unit]
Description=NGINX Server
After=syslog.target network-online.target
Wants=network-online.target
[Service]
Type=forking
PIDFile=/run/nginx.pid
ExecStartPre=/usr/local/nginx/sbin/nginx -t
ExecStart=/usr/local/nginx/sbin/nginx
ExecReload=/usr/local/nginx/sbin/nginx -s reload
ExecStop=/bin/kill -s QUIT $MAINPID
PrivateTmp=true
[Install]
WantedBy=multi-user.target
```

Figure 1.3: Default systemd service file for Nginx

Note

Make sure your paths are correct, as well as the `After=` part which tells `systemd` to execute NGINX only after `syslog.target` and `network-online.target`. Add your own services there, such as your database server, your PHP server, and more.

2. Once your file has been saved, reload the `systemd` configuration using `systemctl daemon-reload`. Then, use `systemctl start nginx` to start the service. You can start, stop, and restart the service in the same way. To enable the service at boot, run the following:

```
systemctl enable nginx
```

3. To stop the NGINX server from booting automatically, you can use `disable` instead of `enable` in the preceding command.
4. You can check whether your NGINX server will boot by running the `is-enabled` command:

```
root@server:~# systemctl is-enabled nginx
enabled
```

Handling system errors

Sometimes, you will encounter errors when starting NGINX with `systemd`. Here is an example:

```
root@server:~# systemctl start nginx
Job for nginx.service failed because the control process exited with error code.
See "systemctl status nginx.service" and "journalctl -xeu nginx.service" for details.
root@server:~# systemctl status nginx
× nginx.service - A high performance web server and a reverse proxy server
     Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
     Active: failed (Result: exit-code) since Mon 2023-09-11 13:27:37 UTC; 11s ago
       Docs: man:nginx(8)
    Process: 6948 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master_process on; (code=exited, status=1/FAILURE)

Sep 11 13:27:37 server systemd[1]: Starting A high performance web server and a reverse proxy server...
Sep 11 13:27:37 server nginx[6948]: nginx: [emerg] unknown directive "Check_your_config_:)" in /etc/nginx/nginx.conf:2
Sep 11 13:27:37 server nginx[6948]: nginx: configuration file /etc/nginx/nginx.conf test failed
Sep 11 13:27:37 server systemd[1]: nginx.service: Control process exited, code=exited, status=1/FAILURE
Sep 11 13:27:37 server systemd[1]: nginx.service: Failed with result 'exit-code'.
Sep 11 13:27:37 server systemd[1]: Failed to start A high performance web server and a reverse proxy server.
```

Figure 1.4: Debugging Nginx through systemd status

In this case, you check the current status using `systemctl status nginx`. Most of the time, this tells you exactly why your NGINX server won't start: very often, the NGINX configuration file is incorrect because of typos or forgotten characters. Make sure to double-check your configuration before restarting the NGINX server.

We have been diving into the compiling options for NGINX as well as its controlling daemon. Should you need professional support from the experts who have made NGINX what it is today, NGINX Plus might be for you.

A quick overview of the possibilities offered by NGINX Plus

As of mid-2013, NGINX, Inc., the company behind the NGINX project, also offers a paid subscription called NGINX Plus. The announcement came as a surprise for the open source community but several companies quickly jumped on the bandwagon and reported amazing improvements in terms of performance and scalability.

NGINX, Inc., the high-performance web company, today announced the availability of NGINX Plus, a fully-supported version of the popular NGINX open source software complete with advanced features and offered with professional services. The product is developed and supported by the core engineering team at *Nginx Inc.*, and is available immediately on a subscription basis.

As business requirements continue to evolve rapidly, such as the shift to mobile and the explosion of dynamic content on the Web, CIOs are continuously looking for opportunities to increase application performance and development agility, while reducing dependencies on their infrastructure. NGINX Plus provides a flexible, scalable, uniformly applicable solution that was purpose built for these modern, distributed application architectures.

Considering the pricing plans ($1,500 per year per instance) and the additional features made available, this platform is indeed clearly aimed at large corporations looking to integrate NGINX into their global architecture seamlessly and effortlessly. Professional support from the NGINX team is included and discounts can be offered for multiple-instance subscriptions. This book covers the open source version of NGINX only and does not detail the advanced functionality offered by NGINX Plus. For more information about the paid subscription, take a look at `https://www.nginx.com/`.

Summary

This chapter covered a number of critical steps. We first made sure that your system contained all the required components for compiling NGINX. We then proceeded to select the proper version branch for your usage – will you be using the stable version or a more advanced yet potentially less stable one? After downloading the source and configuring the compilation process by enabling or disabling features and modules such as SSL, GeoIP, and more, we compiled the application and installed it on the system in the directory of your choice. We created a `unit service` file and modified the system boot sequence to schedule the service to be started.

From this point on, NGINX is installed on your server and automatically starts with the system. Your web server is functional, though it does not yet answer the most basic functionality – serving a website. The first step toward hosting a website will be to prepare a suitable configuration file. The next chapter will cover the basic configuration of NGINX and will teach you how to optimize performance based on the expected audience and system resources.

2
Basic NGINX Configuration

In this chapter, we will begin to establish an appropriate configuration for your web server. For this purpose, we need to first discover the syntax that is used in the NGINX configuration files. Then, we need to understand the various directives that will let you prepare and optimize your web server for different traffic patterns and hardware setups. Finally, we will create some test pages to make sure that everything has been done correctly and that the configuration is valid. We will only approach basic configuration directives here. This chapter and the next ones will detail more advanced topics, such as HTTP module configuration and usage, creating virtual hosts, and setting up SSL.

This chapter covers the following topics:

- The presentation of the configuration syntax
- Basic module directives
- Testing and maintaining your web server

Delving into the configuration file syntax

A configuration file is generally a text file that is edited by the administrator and parsed by a program. By specifying a set of values, you define the behavior of the program. In Linux-based operating systems, the majority of applications rely on vast, complex configuration files that often turn out to be a nightmare to manage. Apache, Postfix, and Bind – all of these names bring up bad memories in the mind of a Linux system administrator.

The fact is that all of these applications use their own configuration files with different syntaxes and styles. PHP works with a Windows-style `.ini` file, Sendmail uses the M4 macro-processor to compile configuration files, Zabbix pulls its configuration from a MySQL database, and so on. There is, unfortunately, no well-established standard, and the same applies to NGINX – you will be required to study a new syntax with its own particularities and its own vocabulary:

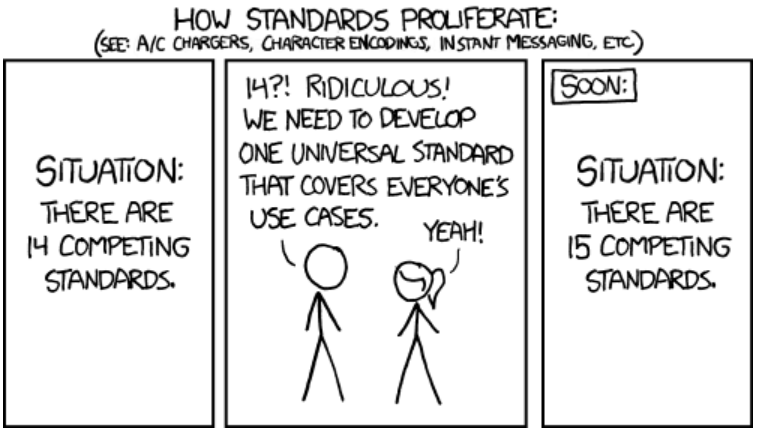


Figure 2.1: A famous website among tech enthusiasts that makes fun of the situation regarding standards

Note

Why isn't there a universal standard for configuration file syntax? A possible explanation is provided by *Randall Munroe* at `https://xkcd.com/` (reproduced with authorization).

Conversely (and this is one of its advantages), configuring NGINX turns out to be rather simple, at least in comparison to Apache or other mainstream web servers. There are only a few mechanisms that need to be mastered – *directives*, *blocks*, and the *overall logical structure* (which will be covered in *Chapter 3*). Most of the actual configuration process will consist of writing values for directives.

Configuration directives

The NGINX configuration file can be described as a list of directives organized in a logical structure. The entire behavior of the application is defined by the values that you give to those directives.

By default, NGINX makes use of one main configuration file. The path of this file was defined in the steps described in *Chapter 1*, in the *Build configuration issues* section. If you did not edit the configuration file path and prefix options, they should be located at `/usr/local/nginx/conf/nginx.conf`.

However, if you installed NGINX with a package manager, your configuration file will likely be located at `/etc/nginx/nginx.conf`, and the contents of the file may be quite different from the version that comes in the original NGINX source code package. Now, let's take a quick peek at the first few lines of this initial setup:

```
  GNU nano 6.2                    /etc/nginx/nginx.conf *
user www-data;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;
events {
        worker_connections 1024;
        # multi_accept on;
}
http {
        ##
        # Basic Settings
        ##
        sendfile on;
        tcp_nopush on;
        types_hash_max_size 2048;
        # server_tokens off;

        # server_names_hash_bucket_size 64;
```

Figure 2.2: nano is used to edit the nginx.conf file

> **Note**
> A default configuration file is bundled with the `Nginx` source code package.

Let's take a closer look at the first two lines:

```
#user nobody;
worker_processes 1;
```

As you can probably make out from the # character, the first line is a comment. In other words, it is a piece of text that is not interpreted and has no value whatsoever. Its sole purpose is to be read by whoever opens the file, or to temporarily disable parts of an existing configuration section. You can use the # character at the beginning of a line or after a directive.

The second line is an actual statement – a directive. The first bit (`worker_processes`) represents a setting key, to which you append one or more values. In this case, the value is `1`, indicating that NGINX should function with a single worker process (more information about this particular directive is given in further sections).

> **Note**
> Directives always end with a semicolon (`;`).

Each directive has a unique meaning and defines a particular feature of an application. It may also have a particular syntax. For example, the `worker_process` directive only accepts one numeric value, whereas the `user` directive lets you specify up to two character strings, one for the user account (that the NGINX worker processes should run as) and a second for the `user` group.

NGINX works in a modular way, and as such, each module comes with a specific set of directives. The most fundamental directives are part of the NGINX core module and will be detailed in this chapter. As for other directives brought in by other modules, they will be explored in the later chapters.

Organization and inclusions

In *Figure 2.2*, you may have noticed a particular directive – `include`:

```
include mime.types;
```

As the name suggests, this directive will perform an inclusion of the specified file. In other words, the contents of the file will be inserted at this exact location. Here is a practical example that will help you understand:

- `nginx.conf`:

```
user nginx nginx;
worker_processes 4;
include other_settings.conf;
```

- `other_settings.conf`:

```
error_log logs/error.log;
pid logs/nginx.pid;
The final result, as interpreted by Nginx, is as follows:
user nginx nginx;
worker_processes 4;
error_log logs/error.log;
pid logs/nginx.pid;
```

Inclusions are processed recursively. In this case, you have the possibility to use the `include` directive again in the `other_settings.conf` file, in order to include yet another file.

In the initial configuration setup, there are two files in use – `nginx.conf` and `mime.types`. However, in the case of a more advanced configuration, there may be five or more files, as described in the following table:

Standard name	Description
`nginx.conf`	The base configuration of the application.
`mime.types`	A list of file extensions and their associated MIME types.
`fastcgi.conf`	A FastCGI-related configuration.
`proxy.conf`	A proxy-related configuration.
`sites.conf`	The configuration of the websites served by NGINX, also known as **virtual hosts**. It's recommended to create separate files for each domain.

Table 2.1: A table detailing the configuration files included by default

These filenames were defined conventionally; nothing actually prevents you from regrouping your FastCGI and proxy settings into a common file named `proxy_and_fastcgi_config.conf`.

The `include` directive supports filename globbing – in other words, filenames referenced with the `*` wildcard, where `*` can match zero, one, or more consecutive characters:

```
include sites/*.conf;
```

This will include all files with a name that ends with `.conf` in the `sites` folder. This mechanism allows you to create a separate file for each of your websites and include them all at once.

Be careful when including a file. If the specified file does not exist, the configuration checks will fail and NGINX will not start:

```
[root@server nginx]# ./nginx -t
[emerg]: open() "/etc/nginx/dummyfile.conf" failed (2: No such file or
directory) in /etc/nginx/nginx.conf:48
```

The previous statement is not true for inclusions with wildcards. Moreover, if you insert `include dummy*.conf` into your configuration and test it (whether there is any file matching this pattern on your system or not), here is what should happen:

```
[root@server nginx]# ./nginx -t
the configuration file /etc/nginx/nginx.conf syntax is ok
configuration file /etc/nginx/nginx.conf test is successful
```

Directive blocks

Directives are brought in by modules; if you activate a new module, a specific set of directives becomes available. Modules can also enable **directive blocks**, which allow for a logical construction of the configuration:

```
events {
    worker_connections 1024;
}
```

The `events` block that you can find in the default configuration file is brought in by the `events` module. The directives that the module enables can only be used within that block. In the preceding example, `worker_connections` will only make sense in the context of the `events` block. Conversely, some directives must be placed at the root of the configuration file because they have a global effect on the server. The root of the configuration file is also known as the **main block**.

For the most part, blocks can be nested into each other, following a specific logic. The following sequence demonstrates the structure of a simple website setup, making use of nested blocks:

```
http {
    server {
        listen 80;
        server_name example.com;
        access_log /var/log/nginx/example.com.log;
        location ^~ /admin/ {
            index index.php;
       }
    }
}
```

The topmost directive block is the `http` block, in which you may declare a variety of configuration directives, as well as one or more `server` blocks. A `server` block allows you to configure a virtual host – in other words, a website that is to be hosted on your machine. The `server` block, in this example, contains some configuration that applies to all HTTP requests with a `Host` header exactly matching `example.com`.

Within this `server` block, you may insert one or more `location` blocks. These allow you to enable settings only when the requested URI matches the specified path. More information is provided in the *The Location block* section of *Chapter 3*.

Last but not least, configuration is inherited within children blocks. The `access_log` directive (defined at the `server`-block level in this example) specifies that all HTTP requests for this server should be logged into a text file. This is still true within the `location` child block, although you have the possibility of disabling it by reusing the `access_log` directive:

```
[...]
    location ^~ /admin/ {
        index index.php;
        access_log off;
    }
[...]
```

In this case, logging will be enabled everywhere on the website, except for the `/admin/` location path. The value set for the `access_log` directive at the `server`-block level is overridden by the one at the `location`-block level.

Advanced language rules

There are a number of important observations regarding the NGINX configuration file syntax. These will help you understand certain language rules that may seem confusing if you have never worked with NGINX before.

Directives accept specific syntaxes

You may indeed stumble upon complex syntaxes that can be confusing at first sight:

```
rewrite ^/(.*)\.(png|jpg|gif)$ /image.php? file=$1&format=$2 last;
```

Syntaxes are directive-specific. While the `root` directive only accepts a simple character string, defining the folder containing files that should be served for a website, the `location` block or the `rewrite` directive supports complex expressions in order to match particular patterns. Some other directives, such as `listen`, accept up to *17* different parameters. Syntaxes will be explained along with directives in their respective chapters.

Later on, we will detail a module (the *rewrite* module) that allows for a much more advanced logical structure through the `if`, `set`, `break`, and `return` blocks and directives, as well as the use of variables. With all of these new elements, configuration files will begin to look like programming scripts. Anyhow, the more modules we discover, the richer the syntax becomes.

Diminutives in directive values

Finally, you cany use the following diminutives to specify a file size in the context of a directive value:

- **k or K**: **Kilobytes**
- **m or M**: **Megabytes**
- **g or G**: **Gigabytes**

As a result, the following three syntaxes are correct and equal:

```
client_max_body_size 2G;
client_max_body_size 2048M;
client_max_body_size 2097152k;
```

NGINX does not allow you to insert the same directive more than once within the same block (although there are a few exceptions, such as `allow` or `deny`); should you do so, the configuration will be considered invalid, and NGINX will refuse to start up or reload.

Additionally, when specifying a time value, you can use the following shortcuts:

- **ms**: **Milliseconds**
- **s**: **Seconds**

- **m**: **Minutes**
- **h**: **Hours**
- **d**: **Days**
- **w**: **Weeks**
- **M**: **Months** (30 days)
- **y**: **Years** (365 days)

This becomes especially useful in the case of directives that accept a period of time as a value:

```
client_body_timeout 3m;
client_body_timeout 180s;
client_body_timeout 180;
```

The default time unit is seconds; the last two preceding lines thus result in identical behavior. It is also possible to combine two values with different units:

```
client_body_timeout 1m30s;
client_body_timeout '1m 30s 500ms';
```

The latter variant is enclosed in quotes, since values are separated by spaces.

Variables

Modules also provide variables that can be used in the definition of directive values. For example, the NGINX HTTP core module defines the `$nginx_version` variable. Variables in NGINX always start with `$` – the dollar sign. When setting the `log_format` directive, you can include all kinds of variables in the format string:

```
[...]
location ^~ /admin/ {
    access_log logs/main.log;
    log_format main '$pid - $nginx_version - $remote_addr';
}
[...]
```

Some directives do not allow you to use variables:

```
error_log logs/error-$nginx_version.log;
```

The preceding directive is valid, syntax-wise. However, it simply generates a file named `error-$nginx_version.log`, without parsing the variable.

String values

Character strings that you use as directive values can be written in three forms. First, you can enter a value without quotes:

```
root /home/example.com/www;
```

However, if you want to use a particular character, such as a blank space ( ), a semicolon (`;`), or a pair of curly braces (`{ }`), you will need to either prefix said character with a backslash (`\`) or enclose the entire value in single or double quotes:

```
root '/home/example.com/my web pages';
```

NGINX makes no difference whether you use single or double quotes. Note that variables inserted in strings within quotes will be expanded normally unless you prefix the `$` character with a backslash (`\`).

We now have a better understanding of how the configuration works, as we've learned about the syntax and the diminutives. We are now ready to cover the NGINX modules in depth, allowing us to fine-tune NGINX to take full advantage of the hardware and network to better cover our needs.

Looking at the base module directives

In this section, we will take a closer look at the base modules. We are particularly interested in answering two questions – *what are base modules?* and *what directives are made available?*

What are base modules?

The base modules offer directives that allow you to define the parameters of the basic functionality of NGINX. They cannot be disabled at compile time, and as a result, the directives and blocks they offer are always available. Three base modules are distinguished:

- **Core module**: This has essential features and directives, such as process management and security
- **Events module**: This lets you configure the inner mechanisms of the networking capabilities
- **Configuration module**: This enables the inclusion mechanism

These modules offer a large range of directives; we will detail them individually, with their syntaxes and default values.

NGINX process architecture

Before we start detailing the basic configuration directives, it is necessary to understand the overall process architecture – that is, how the NGINX daemon works behind the scenes. Although the application comes as a simple binary file (and a somewhat lightweight background process), the way it functions at runtime can be relatively complex.

At the very moment of starting NGINX, one unique process exists in memory – `master process`. It is launched with the current user and group permissions, usually `root/root` if the service is launched at boot time by an `init` script. The master process itself does not process any client requests; instead, it spawns processes that do – *worker processes*, which are affected by a customizable user and group.

From the configuration file, you are able to define the number of worker processes, the maximum connections per worker process, the user and group the worker processes run under, and so on. The following screenshot shows an example of a running instance of NGINX, with eight worker processes running under the `www-data` user account:

```
root@server:~# ps faux | grep nginx
root        3668  0.0  0.0   7004  2220 pts/1    S+   16:11   0:00  \_ grep --color=auto nginx
root        3657  0.0  0.0  55212  1680 ?        Ss   16:11   0:00 nginx: master process /usr/sbin/nginx
www-data    3658  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3659  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3660  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3661  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3662  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3663  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3664  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
www-data    3665  0.0  0.0  55844  5516 ?        S    16:11   0:00  \_ nginx: worker process
root@server:~# _
```

Figure 2.3: ps faux is used to list the processes and grep is used to show only nginx

Core module directives

The following is the list of directives made available by the core module. Most of these directives must be placed at the root of the configuration file and can only be used once. However, some of them are valid in multiple contexts.

If that is the case, the following is the list of valid contexts under the directive name:

Name and context	Syntax and description
`daemon`	**Accepted values**: `on` or `off`. **Syntax**: `daemon on;`. **Default value**: `on`. Enables or disables daemon mode. If you disable it, the program will not be started in the background; it will stay in the foreground when launched from the shell. This may come in handy for debugging, in situations where you need to know what causes NGINX to crash, and when.

Name and context	Syntax and description
`debug_points`	**Accepted values**: `stop` or `abort`. **Syntax**: `debug_points stop;`. **Default value**: None. Activates debug points in NGINX. Use `stop` to interrupt the application when a debug point comes about in order to attach a debugger. Use `abort` to abort the debug point and create a core dump file. To disable this feature, simply do not use the directive.
`Env`	**Syntax**: `env MY_VARIABLE; env MY_VARIABLE=my_value;`. Allows you to define or redefine environment variables
`error_log` **Context**: `main`, `http`, `server`, and `location`	**Syntax**: `error_log /file/path level;`. **Default value**: `logs/error.log error`. **Where level is one of these values**: `debug`, `info`, `notice`, `warn`, `error`, `crit`, `alert`, and `emerg` (from most to least detailed – `debug` provides frequent log entries, while `emerg` only reports the most critical errors). **Enables error logging at different levels**: The application, HTTP server, virtual host, and virtual host directory. By redirecting the log output to `/dev/null`, you can disable error logging. Use the following directive at the root of the configuration file: `error_log /dev/null crit;` Instead of specifying a file path, you might also select one of the following alternatives – `stderr` will send log entries to the standard error file, and `Syslog` will send them to the system log. These alternatives are further detailed in *Chapter 11*.
`lock_file`	**Syntax (file path)**: `lock_file logs/nginx.lock;`. **Default value**: Defined at compile time. Use a lock file for mutual exclusion. This is disabled by default, unless you enabled it at compile time. On most operating systems, the locks are implemented using atomic operations, so this directive is ignored anyway.

Name and context	Syntax and description
`load_module` **Context**: `main`	**Syntax (file path)**: `load_module modules/ngx_http_geoip_module.so;`. **Default value**: None. Load a dynamically compiled module at runtime
`log_not_found` **Context**: `main`, `http`, `server`, and `location`	**Accepted values**: `on` or `off`. **Syntax**: `log_not_found on;`. **Default value**: `on`. Enables or disables logging of `404` not found HTTP errors. If your logs get filled with `404` errors due to missing `favicon.ico` or `robots.txt` files, you might want to turn this off.
`master_process`	**Accepted values**: `on` or `off`. **Syntax**: `master_process on;`. **Default value**: `on`. If enabled, NGINX will start multiple processes – the main process (the master process) and worker processes. If disabled, NGINX works with a unique process. This directive should be used for testing purposes only, as it disables the master process; clients thus cannot connect to your server.
`pcre_jit`	**Accepted values**: `on` or `off`. **Syntax**: `pcre_jit on;`. Enables or disables **Just-in-Time** (**JIT**) compilation for regular expressions (PCRE from version 8.20 and above), which may speed up their processing significantly. For this to work, the PCRE libraries on your system must be specifically built with the `--enable-jit` configuration argument. When configuring your NGINX build, you must also add the `--with-pcre-jit` argument.
`Pid`	**Syntax (file path)**: `pid logs/nginx.pid;`. **Default value**: Defined at compile time. The path of the `pid` file for the NGINX daemon. The default value can be configured at compile time. Make sure to enable this directive and set its value properly, since the `pid` file can be used by the NGINX `init` script, depending on your operating system.

Name and context	Syntax and description
`ssl_engine`	**Syntax (character string)**: `ssl_engine enginename;`. **Default value**: None. Where `enginename` is the name of an available hardware SSL accelerator on your system. To check for available hardware SSL accelerators, run this command from the shell: `openssl engine -t`
`thread_pool`	**Syntax**: `thread_pool name threads=number [max_queue=number];`. **Default value**: `thread_pool default threads=32 max_queue=65536;`. Defines a thread pool reference that can be used with the `aio` directive, in order to serve larger files asynchronously. Further details are provided in *Chapter 9*.
`timer_resolution`	**Syntax (numeric (time))**: `timer_resolution 100ms;`. **Default value**: None. Controls the interval between system calls to `gettimeofday()` to synchronize the internal clock. If this value is not specified, the clock is refreshed after each kernel event notification.
`user`	**Syntax**: `user username groupname;` `user username;`. **Default value**: Defined at compile time. If still undefined, the user and group of the NGINX master process are used. Allows you to define the user account and, optionally, the user group used to start the NGINX worker processes. For security reasons, you should make sure to specify a user and group with limited privileges. For example, create a new user and group dedicated to NGINX, and remember to apply proper permissions on the files that will be served.

Name and context	Syntax and description

`worker_cpu_affinity`	**Syntax**:
	`worker_cpu_affinity 1000 0100 0010 0001;`
	`worker_cpu_affinity 10 10 01 01;`
	`worker_cpu_affinity auto;.`
	Default value: None.
	This directive works in conjunction with `worker_processes`. It lets you affect worker processes to CPU cores.
	There are as many series of digit blocks as worker processes; there are as many digits in a block as your CPU has cores.
	If you configure NGINX to use three worker processes, there are three blocks of digits. For a dual-core CPU, each block has two digits:
	`worker_cpu_affinity 01 01 10;`
	The first block (`01`) indicates that the first worker process should be effected to the second core.
	The second block (`01`) indicates that the second worker process should be affected to the second core.
	The third block (`10`) indicates that the third worker process should be affected to the first core.
	The auto value allows NGINX to automatically manage the process binding. This differs from the default of not being specified, which means the OS will manage it.
	Note that affinity is only recommended for multi-core CPUs, not for processors with hyperthreading or similar technologies.

`worker_priority`	**Syntax (numeric)**: `worker_priority 0;`. **Default value**: `0`. Defines the priority of the worker processes, from `-20` (highest) to `19` (lowest). The default value is `0`. Note that kernel processes run at priority level `-5`, so it's not recommended that you set the priority to `-5` or less.

Name and context	Syntax and description
`worker_processes`	**Syntax (numeric or auto)**: `worker_processes 4;`. **Default value**: `1`. Defines the number of worker processes. NGINX offers to separate the treatment of requests into multiple processes. The default value is `1`, but it's recommended to increase this value if your CPU has more than one core. Besides, if a process gets blocked due to slow I/O operations, incoming requests can be delegated to the other worker processes. Alternatively, you may use the `auto` value, which will let NGINX select an appropriate value for this directive. By default, it is the amount of CPU cores detected on the system.
`worker_rlimit_core`	**Syntax (numeric (size))**: `worker_rlimit_core 100m;`. **Default value**: None. Defines the size of core files per worker process
`worker_rlimit_nofile`	**Syntax (numeric)**: `worker_rlimit_nofile 10000;`. **Default value**: None. Defines the number of files a worker process can use simultaneously.
`working_directory`	**Syntax (directory path)**: `working_directory /usr/local/nginx/;`. **Default value**: The prefix switch defined at compile time. Working directory used for worker processes; it is only used to define the location of core files. The worker process user account (the `user` directive) must have write permissions on this folder in order to be able to write core files.
`worker_aio_requests`	**Syntax (numeric)**: `worker_aio_requests 10000;`. If you are using `aio` with the `epoll` connection processing method, this directive sets the maximum number of outstanding asynchronous I/O operations for a single worker process.
`worker_shutdown_timeout`	**Syntax (time)**: `worker_shutdown_timeout 5s;`. Configures the time limit for the graceful shutdown of worker processes. If exceeded, NGINX will try to forcefully close workers to complete the shutdown.

Table 2.2: A table detailing the core module directives

The events module

The `events` module comes with directives that allow you to configure network mechanisms. Some of the parameters have an important impact on an application's performance.

All of the directives listed in the following table must be placed in the events block, which is located at the root of the configuration file:

```
user nginx nginx;
master_process on;
worker_processes 4;
events {
  worker_connections 1024;
  use epoll;
}
[...]
```

These directives cannot be placed elsewhere (if you do so, the configuration test will fail):

Directive name	Syntax and description
`accept_mutex`	**Accepted values**: `on` or `off`. **Syntax**: `accept_mutex on;` . **Default value**: As of version 1.11.3, `off`; prior to version 1.11.3, `on`. Enables or disables the use of an accept mutex (mutual exclusion) to open listening sockets.
`accept_mutex_delay`	**Syntax (numeric (time))**: `accept_mutex_delay 500ms;`. **Default value**: 500 milliseconds. Defines the amount of time a worker process should wait before trying to acquire the resource again. This value is not used if the `accept_mutex` directive is set to `off`.
`debug_connection`	**Syntax (IP address or CIDR block)**: `debug_connection  172.63.155.21;` `debug_connection  172.63.155.0/24;`. **Default value**: None. Writes detailed logs for clients matching this IP address or address block. The debug information is stored in the file specified with the `error_log` directive, enabled with the debug level. NGINX must be compiled with the `--debug` switch in order to enable this feature.

Directive name	Syntax and description
`multi_accept`	**Syntax (on or off)**: `multi_accept off;`. **Default value**: `off`. Defines whether or not NGINX should accept all incoming connections at once from the listening queue.
`use`	**Accepted values**: `/dev/poll`, `epoll`, `eventport`, `kqueue`, `rtsig`, or `select`. **Syntax**: `use kqueue;`. **Default value**: Defined at compile time. Selects the `event` model among the available ones (the ones that you enabled at compile time). NGINX automatically selects the most appropriate one, so you should not have to modify this value. The supported models are as follows: `select`: The default and standard module, which is used if the OS does not support a more efficient one (it's the only available method under Windows). This method is not recommended for servers that expect to be under high load. `poll`: This is automatically preferred over `select` but is not available on all systems. `kqueue`: An efficient method for FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0, and macOS OSes. `epoll`: An efficient method for Linux 2.6+ based OSes. `rtsig`: Real-time signals, available as of Linux 2.2.19 but unsuitable for high-traffic profiles, as default system settings only allow 1,024 queued signals. `/dev/poll`: An efficient method for Solaris 7 11/99+, HP/UX 11.22+, IRIX 6.5.15+, and Tru64 UNIX 5.1A+ OSes. `eventport`: An efficient method for Solaris 10, although a security patch is required.
`worker_connections`	**Syntax (numeric)**: `worker_connections 1024;`. **Default value**: None. Defines the number of connections that a worker process may treat simultaneously.

Table 2.3: A table detailing the events module directives

Configuration module

The NGINX `configuration` module is a simple module that enables file inclusions with the `include` directive, as previously described in the *Organization and inclusions* section. The directive can be inserted anywhere in the configuration file and accepts a single parameter – a file path:

```
include /file/path.conf;
include sites/*.conf;
```

> **Note**
>
> If you do not specify an absolute path, the file path is relative to the configuration directory. By default, `include sites/example.conf` will include the
>
> `/usr/local/nginx/conf/sites/example.conf` file.

Necessary adjustments

Several core directives deserve to be adjusted carefully upon preparing the initial setup of NGINX on your server. We will review several of these directives and the possible values you can set:

- `user root;`: This directive specifies that the worker processes will be started as `root`. It is dangerous for security, as it grants NGINX full permissions over your filesystem. You need to create a new user account on your system and make use of it here. The recommended value (granted that a `www-data` user account and group exist on the system) is `user www-data www-data;`.
- `worker_processes 1;`: With this setting, only one worker process will be started, which implies that all requests will be processed by a unique execution flow. This also implies that the execution is delegated to only one core of your CPU. It is highly recommended to increase this value; you should have at least one process per CPU core. Alternatively, just set this to `auto` to leave it up to NGINX to determine the optimal value. The recommended value is `worker_processes auto;`.
- `worker_priority 0;`: By default, the worker processes are started with a regular priority. If your system performs other tasks simultaneously, you might want to grant a higher priority to the NGINX worker processes. In this case, you should decrease the value; *the smaller the value, the higher the priority*. Values range from `-20` (the highest priority) to `19` (the lowest priority). There is no recommended value here, as it completely depends on your situation. However, you should not set it to under `-5`, as it is the default priority for kernel processes.

- `log_not_found on;`: This directive specifies whether NGINX should log `404 errors` or not. While these errors may, of course, provide useful information about missing resources, a lot of them may be generated by web browsers trying to reach the *favicon* (the conventional `/favicon.ico` of a website) or robots trying to access the indexing instructions (`robots.txt`). Set this to `off` if you want to ensure your log files don›t get cluttered by **Error 404** entries, but keep in mind that this could deprive you of potentially important information about other pages that visitors failed to reach. Note that this directive is part of the HTTP core module. Refer to the next chapter for more information.
- `worker_connections 1024;`: This setting, combined with the number of worker processes, allows you to define the total number of connections accepted by the server simultaneously. If you enable four worker processes, each accepting 1,024 connections, your server will treat a total of 4,096 simultaneous connections. You need to adjust this setting to match your hardware – the more RAM and CPU power your server relies on, the more connections you can accept concurrently. If your server is a huge monster meant to host high-traffic sites, you will want to increase this value.

We've taken a look at the configuration. Next, we are going to put this into practice by trying our configuration and launching a test server to see how NGINX behaves.

Testing your server

At this point, you have configured several basic directives that affect the core functioning of NGINX. We will perform a simple test to ensure that all is working as expected and that you are ready to further configure and deploy your websites.

Creating a test server

In order to perform simple tests, such as connecting to a server with a web browser, we need to set up a website for NGINX to serve. A test page comes with the default package in the `html` folder (`/usr/local/nginx/html/index.html`), and the original `nginx.conf` is configured to serve this page. Here is the section that we are interested in for now:

```
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       80;
        server_name  localhost;
```

```
        location / {
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
}
```

As you can perhaps already tell, this segment configures NGINX to serve a website:

- It opens a listening socket on port `80`
- Accessible at the `http://localhost/` address
- With the `index.html` index page

For more details about these directives, refer to the *HTTP module configuration* section in *Chapter 3*. Anyhow, fire up your favorite web browser and visit `http://localhost/`:

Figure 2.4: The default page for the NGINX server

You should be greeted with a welcome message; if you aren't, then check the configuration again and make sure you reloaded NGINX in order to apply the changes.

Upgrading NGINX gracefully

There are many situations where you will need to replace the NGINX binary – for example, when you compile a new version and wish to put it in production, or simply after having enabled new modules and rebuilt the application. What most administrators would do in this situation is stop the server, copy the new binary over the old one, and restart NGINX. While this is not considered to be a problem for most websites, there may be some cases where uptime is critical and connection losses should be avoided at all costs.

Fortunately, NGINX embeds a mechanism that allows you to switch binaries with uninterrupted uptime; zero percent request loss is guaranteed if you follow these steps carefully:

1. Replace the old NGINX binary (by default, `/usr/local/nginx/sbin/nginx`) with the new one.
2. Find `pid` of the NGINX master process – for example, with `ps x|grep nginx|grep master` or by looking at the value found in the `.pid` file.
3. Send a `USR2` (`12`) signal to the master process, `kill -USR2 1234`, replacing `1234` with the `pid` found in *step 2*. This will initiate the upgrade by renaming the old `.pid` file and running the new binary.
4. Send a `WINCH` (`28`) signal to the old master process, `kill -WINCH 1234`, replacing `1234` with the `pid` found in *step 2*. This will engage a graceful shutdown of the old worker processes.
5. Make sure that all of the old worker processes are terminated, and then send a `QUIT` signal to the old master process, `kill -QUIT 1234`, replacing `1234` with the `pid` found in *step 2*.

Congratulations! You have successfully upgraded NGINX and have not lost a single connection.

Summary

This chapter provided a first approach to the configuration architecture by studying the syntax and the core module directives that have an impact on the overall server performance. We then went through a series of adjustments in order to fit your own profile, followed by a procedure to upgrade your running NGINX server without losing connections.

However, this is just the beginning. Practically everything that we will do from now on is to prepare configuration sections. The next chapter will detail more advanced directives by further exploring the module system and the exciting possibilities that are offered to you by it.

Part 2: Dive into NGINX

In this part, you'll deepen your knowledge of NGINX by exploring its various modules and advanced configuration options. In this part, you'll delve into the intricacies of basic modules such as the HTTP module and the rewriting module, as well as integration with PHP and Python. You'll also learn how to adapt NGINX to your specific needs.

This part includes the following chapters:

- *Chapter 3, Exploring the HTTP Configuration*
- *Chapter 4, Exploring Module Configuration in NGINX*
- *Chapter 5, PHP and Python with NGINX*
- *Chapter 6, NGINX as a Reverse Proxy*

3
Exploring the HTTP Configuration

At this stage, we have a working NGINX setup—not only is it installed on the system and launched automatically on startup but it's also organized and optimized with the help of basic directives. It's now time to go one step further into the configuration by discovering the HTTP core module. This module is an essential component of the HTTP configuration—it allows you to set up websites to be served, also referred to as virtual hosts.

This chapter will cover the following topics:

- An introduction to the HTTP core module and its three new blocks
- Exploring the HTTP core module directives
- Exploring the directives of HTTP/2
- Exploring the variables introduced by the HTTP core module
- Understanding and exploring the `location` block

By the end of this chapter, you will know all the main points about configuring the NGINX server, its modules, and you will be able to host your first website.

An introduction to the HTTP core module and its three new blocks

The **HTTP core module** is the component that contains all of the fundamental blocks, directives, and variables of the HTTP server. It's enabled by default when you configure the build (as described in *Chapter 1*), but it's actually optional—you can decide not to include it in your custom build. Doing so will completely disable all HTTP functionalities, and all of the other HTTP modules will not be compiled. Obviously, if you purchased this book, it's highly likely that you are interested in the web-serving capacities of NGINX, so you will have this enabled.

This module is the largest of all standard NGINX modules—it provides an impressive number of directives and variables. In order to understand all of these new elements and how they come into play, we first need to understand the logical organization introduced by the three main blocks—`http`, `server`, and `location`.

In the previous chapter, we discovered the core module by studying the default NGINX configuration file, which includes a sequence of directives and values with no apparent organization. Then came the `events` module, which introduced the first block, (`events`). This block is the only placeholder for all directives brought in by the `events` module.

As it turns out, the HTTP module introduces three new logical blocks:

- `http`: This block is inserted at the root of the configuration file. It allows you to start defining directives and blocks from all modules related to the HTTP facet of NGINX. Although there is no real purpose in doing so, the block can be inserted multiple times, in which case the directive values inserted in the last block will override the previous ones.
- `server`: This block allows you to declare a website. In other words, a specific website (identified by one or more hostnames—for example, `www.mywebsite.com`) becomes acknowledged by NGINX and receives its own configuration. This block can only be used within the `http` block.
- `location`: This block lets you define a group of settings to be applied to a particular location on a website. This block can be used within a `server` block or nested within another `location` block.

The following diagram summarizes the final structure by providing a couple of basic examples corresponding to actual situations:

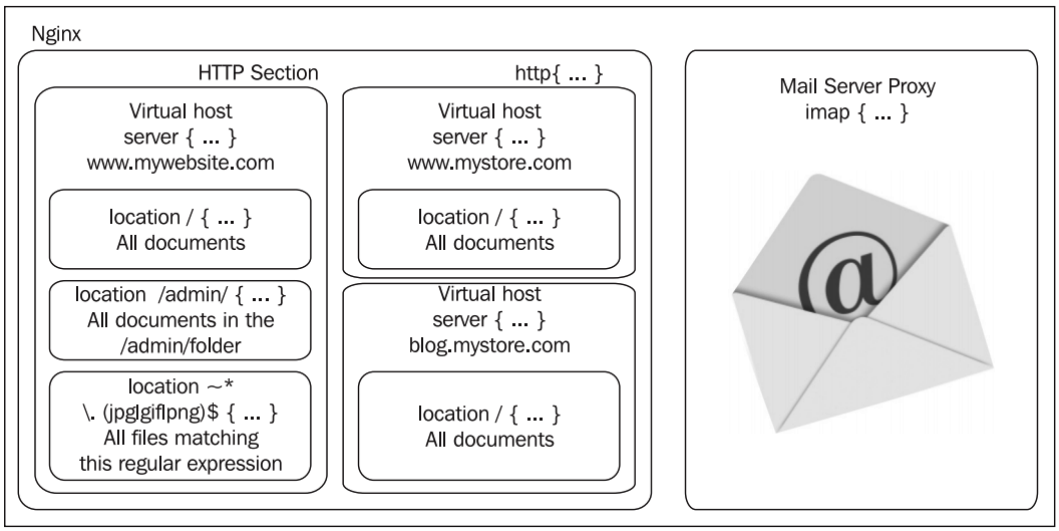


Figure 3.1: Diagram showing block structure and hierarchy with a few examples

The HTTP section, defined by the `http{ . . . }` block, encompasses the entire web-related configuration. It may contain one or more `server{ . . . }` blocks, defining the domains and subdomains that you are hosting. For each of these websites, you have the possibility of defining `location` blocks that let you apply additional settings to a particular request URI, or request URIs matching a pattern.

Remember that the principle of setting inheritance applies here. If you define a setting at the `http{ . . . }` block level (for example, `gzip on` to enable gzip compression), the setting will preserve its value in the potentially incorporated `server` and `location` blocks:

```
http {
   # Enable gzip compression at the http block level
   gzip on;

   server {
      server_name localhost;
       listen 80;

      # At this stage, gzip still set to on

      location /downloads/ {
          gzip off;
     #    This directive only applies to documents found
     #    in /downloads/
       }
   }
}
```

We have learned about the NGINX block structure, and we have put it into practice with a concrete example. We will now cover the modules available in each of these three blocks.

Exploring the HTTP core module directives

At each of the three levels (the blocks discussed previously), directives can be inserted in order to affect the behavior of the web server. The following subsections cover all directives that are introduced by the main HTTP module, grouped thematically. For each directive, an indication regarding the context is given. Some cannot be used at certain levels. For instance, it would make no sense to insert a `server_name` directive at the `http` block level, since `server_name` is a directive directly affecting a virtual host—it should only be inserted in the `server` block. To that extent, the table indicates the possible levels where each directive is allowed—the `http` block, the `server` block, the `location` block, and, additionally, the `if` block.

> **Note**
> This documentation is valid as of Stable version `1.25`. Future updates may alter the *syntax of some directives* or provide new features that are not discussed here.

Socket and host configuration

This set of directives will allow you to configure your virtual hosts, in practice, by creating `server` blocks that you identify either by a hostname or by an IP address and port combination. In addition, some directives will let you fine-tune your network settings by configuring TCP socket options.

listen

Context: `server`

Specifies the IP address and/or the port to be used by the listening socket that will serve the website. Sites are generally served on port `80` (the default value) via HTTP or `443` via HTTPS.

Syntax: `listen [address][:port] [additional options];`

Additional options include the following:

- `default_server`: Specifies that this `server` block is to be used as the default website for any request received at the specified IP address and port
- `ssl`: Specifies that the website should be served using SSL
- `http2`: Enables support for the HTTP/2 protocol, if the `http2` module is present
- `proxy_protocol`: Enables the proxy protocol for all connections accepted on this port
- Other options are related to the `bind` and `listen` system calls:

```
backlog=num, rcvbuf=size, sndbuf=size, accept_filter=filter,
deferred, setfib=number, fastopen=number, ipv6only=on|off,
reuseport, so_keepalive=on|off|[keepidle]:[keepintvl]:[keepcnt],
bind, http2
```

- Here are some examples:

```
listen 192.168.1.1:80; listen 127.0.0.1;
listen 80 default_server;
listen [::a8c9:1234]:80; # IPv6 addresses must be put between
square brackets
listen 443 ssl http2;
```

server_name

Context: `server`

The `server_name` directive assigns one or more hostnames to the `server` block. When NGINX receives an HTTP request, it matches the `Host` header of the request against all `server` blocks. The first `server` block to match this hostname is selected.

If no `server` block matches the desired host, NGINX selects the first `server` block that matches the parameters of the `listen` directive (for example, `listen *:80` would be a catch-all for all requests received on port `80`), giving priority to the first block that has the `default_server` option enabled on the `listen` directive.

> **Important note**
> This directive accepts wildcards as well as regular expressions. In this case, the hostname should start with the ~ character.

Syntax: `server_name hostname1 [hostname2...];`

Examples:

```
server_name www.website.com;
server_name www.website.com website.com;
server_name *.website.com;
server_name .website.com; # combines both *.website.com and website.
com
server_name *.website.*;
server_name ~^(www)\.example\.com$; # $1 = www
```

You may use an empty string as the directive value in order to catch all requests that do not come with a `Host` header, but only after at least one regular name (or _ for a dummy hostname):

```
server_name website.com "";
server_name _ "";
```

server_name_in_redirect

Context: `http`, `server`, and `location`

This directive applies to internal redirects. If set to `on`, NGINX will use the first hostname specified in the `server_name` directive. If set to `off`, NGINX will use the value of the `Host` header from the HTTP request.

Syntax: `on` or `off`

Default value: `off`

server_names_hash_max_size

Context: `http`

NGINX uses hash tables for various data collections (maps, mimes, headers, and so on) in order to speed up the processing of requests. This directive defines the maximum size of the server names hash table. The default value should fit with most configurations. If this needs to be changed, NGINX will automatically tell you on startup or when you reload its configuration.

Syntax: Numeric value

Default value: `512`

server_names_hash_bucket_size

Context: `http`

Sets the bucket size for the server names hash table. Similarly, you should only change this value if NGINX tells you to.

Syntax: Numeric value

Default value: `32` (or `64` or `128`, depending on your processor cache specifications).

port_in_redirect

Context: `http`, `server`, and `location`

If disabled, redirects issued by NGINX will be relative.

Syntax: `on` or `off`

Default value: `on`

absolute_redirect

Context: `http`, `server`, `location`

In the case of a redirect, this directive defines whether or not NGINX should append the port number to the redirection URL.

Syntax: `on` or `off`

Default value: `on`

sendfile

Context: `http`, `server`, `location`

If this directive is enabled, NGINX will use the `sendfile` kernel call to handle file transmission. If disabled, NGINX will handle the file transfer by itself. Depending on the physical location of the file being transmitted (such as NFS), this option may affect the server's performance.

On Linux, using `sendfile` automatically disables asynchronous I/O. If using FreeBSD, it is possible to combine the use of `aio` and `sendfile`.

Syntax: `on` or `off`

Default value: `off`

sendfile_max_chunk

Context: `http` and `server`

This directive defines a maximum data size to be used for each call to `sendfile` (read the previous entry).

Syntax: Numeric value (size)

Default value: `0`

send_lowat

Context: `http` and `server`

An option allowing you to make use of the `SO_SNDLOWAT` flag for TCP sockets under FreeBSD only. This value defines the minimum number of bytes in the buffer for output operations.

Syntax: Numeric value (size)

Default value: `0`

reset_timedout_connection

Context: `http`, `server`, and `location`

When a client connection times out, its associated information may remain in memory depending on the state it was in. Enabling this directive will erase all memory associated with the connection after it times out.

Syntax: `on` or `off`

Default value: `off`

Paths and documents

This section describes directives that configure the documents that should be served for each website, such as the document root, the site index, error pages, and so on.

root

Context: `http`, `server`, `location`, and `if`. Variables are accepted.

Defines the document root, containing the files you wish to serve to your visitors.

Syntax: Directory path

Default value: `html`

Example:

```
root /home/website.com/public_html;
```

alias

Context: `location`. Variables are accepted.

Syntax: Directory (do not forget the trailing `/`) or file path

`alias` is a directive that you place in a `location` block only. It assigns a different path for NGINX to retrieve documents for a specific request. As an example, consider the following configuration:

```
http {
   server {
       server_name localhost;
       root /var/www/website.com/html;
        location /admin/ {
        alias /var/www/locked/;
      }
   }
}
```

When a request for `http://localhost/` is received, files are served from the `/var/www/website.com/html/` folder. However, if NGINX receives a request for `http://localhost/admin/`, the path used to retrieve the files is `/home/website.com/locked/`. Moreover, the value of the document root directive (`root`) is not altered. This procedure is invisible in the eyes of dynamic scripts.

error_page

Context: `http`, `server`, `location`, and `if`. Variables are accepted.

Allows you to affect URIs to HTTP response code and, optionally, to replace the code with another.

Syntax: `error_page code1 [code2...] [=replacement code] [=@block | URI]`, where the replacement code (denoted by `=code`) is one of `301`, `302`, `303`, `307`, or `308`

Examples:

```
error_page 404 /not_found.html;
error_page 500 501 502 503 504 /server_error.html;
error_page 403 http://website.com/;
error_page 404 @notfound; # jump to a named location block
error_page 404 =200 /index.html; # in case of 404 error, redirect to
index.html with a 200 OK response code
```

if_modified_since

Context: `http`, `server`, and `location`

Defines how NGINX handles the `If-Modified-Since` HTTP header. This header is mostly used by search engine spiders (such as Google web crawling bots). The robot indicates the date and time of the last pass. If the requested file has not been modified since then, the server simply returns a `304 Not Modified` response code with no body.

This directive accepts the following three values:

- `off`: Ignores the `If-Modified-Since` header.
- `exact`: Returns `304 Not Modified` if the date and time specified in the HTTP header are an exact match with the actual requested file modification date. If the file modification date is earlier or later, the file is served normally (`200 OK` response).
- `before`: Returns `304 Not Modified` if the date and time specified in the HTTP header is earlier than, or equal to, the requested file modification date.

Syntax: `if_modified_since off | exact | before`

Default value: `exact`

index

Context: `http`, `server`, `location`. Variables are accepted.

Defines the default page that NGINX will serve if no filename is specified in the request (in other words, the index page). You may specify multiple filenames, and the first file to be found will be served. If none of the specified files are found, NGINX will either attempt to generate an automatic index of the files if the `autoindex` directive is enabled (check the `HTTP Autoindex` module) or return a `403 Forbidden` error page. Optionally, you may insert an absolute filename (such as `/page.html`, based on the document root directory) but only as the last argument of the directive.

Syntax: `index file1 [file2...] [absolute_file];`

Default value: `index.html`

Examples:

```
index index.php index.html index.htm;
index index.php index.php /catchall.php;
```

recursive_error_pages

Context: `http`, `server`, `location`

Sometimes an error page, itself served by the `error_page` directive, may trigger an error; in this case, the `error_page` directive is used again (recursively). This directive enables or disables recursive error pages.

Syntax: `on` or `off`

Default value: `off`

try_files

Context: `server`, `location`. Variables are accepted.

Attempts to serve the specified files (arguments *1* to *N-1*); if none of these files exist, it jumps to the respective named `location` block (last argument) or serves the specified URI.

Syntax: Multiple file paths, followed by a named `location` block or a URI

Example:

```
location / {
    try_files $uri $uri.html $uri.xml @proxy;
}
# the following is a "named location block"
location @proxy {
    proxy_pass 127.0.0.1:8080;
}
```

In this example, NGINX tries to serve files normally. If the request URI does not correspond to any existing file, NGINX appends `.html` to the URI and tries to serve the file again. If it still fails, it tries with `.xml`. Eventually, if all of these possibilities fail, another `location` block (`@proxy`) handles the request.

It is important to note that, except for the final argument, `try_files` will serve the literal file with no internal redirect. That means you *cannot* do a `try_files` directive as follows, as this would result in any file matching `$uri.php` being served with the PHP source code. This would leave a security vulnerability where a user could request a `/config` URI and get the contents of `/config.php`:

```
location / {
    try_files $uri $uri.php @proxy;
```

```
}
# the following is a "named location block"
location @proxy {
    proxy_pass 127.0.0.1:8080;
}
```

> **Note**
>
> You may also specify `$uri/` in the list of values in order to test for the existence of a directory with that name.

Client requests

This section documents the way that NGINX handles client requests. Among other things, you are allowed to configure the keep-alive mechanism behavior and possibly log client requests into files.

keepalive_requests

Context: `http`, `server`, and `location`

Maximum number of requests served over a single keep-alive connection.

Syntax: Numeric value

Default value: `100`

keepalive_timeout

Context: `http`, `server`, and `location`

This directive defines the number of seconds the server will wait before closing a keep-alive connection. The second (optional) parameter is transmitted as the value of `Keep-Alive: timeout= <HTTP response header>`. The intended effect is to let the client browser close the connection itself after this period has elapsed. Note that some browsers ignore this setting. Internet Explorer, for instance, automatically closes the connection after around 60 seconds.

Syntax: `keepalive_timeout time1 [time2];`

Default value: `75`

Examples:

```
keepalive_timeout 75;
keepalive_timeout 75 60;
```

keepalive_disable

Context: `http`, `server`, and `location`

This option allows you to disable the `keepalive` functionality for browser families of your choice.

Syntax: `keepalive_disable browser1 browser2;`

Default value: `msie6`

send_timeout

Context: `http`, `server`, and `location`

The number of times after which NGINX closes an inactive connection. A connection becomes inactive the moment a client stops transmitting data.

Syntax: Time value (in seconds)

Default value: `60`

client_body_in_file_only

Context: `http`, `server`, and `location`

If this directive is enabled, the body of incoming HTTP requests will be stored in actual files on the disk. The client body corresponds to the client HTTP request raw data, minus the headers (in other words, the content transmitted in `POST` requests). Files are stored as plain-text documents.

This directive accepts three values:

- `off`: Do not store the request body in a file
- `clean`: Store the request body in a file and remove the file after a request is processed
- `on`: Store the request body in a file, but do not remove the file after the request is processed (not recommended unless for debugging purposes)

Syntax: `client_body_in_file_only on | clean | off`

Default value: `off`

client_body_in_single_buffer

Context: `http`, `server`, and `location`

Defines whether or not NGINX should store the request body in a single buffer in memory.

Syntax: `on` or `off`

Default value: `off`

client_body_buffer_size

Context: `http`, `server`, and `location`

Specifies the size of the buffer holding the body of client requests. If this size is exceeded, the body (or at least part of it) will be written to the disk. Note that, if the `client_body_in_file_only` directive is enabled, request bodies are always stored into a file on the disk, regardless of their size (whether they fit in the buffer or not).

Syntax: Size value

Default value: `8k` or `16k` (two memory pages) depending on your computer architecture.

client_body_temp_path

Context: `http`, `server`, and `location`

Allows you to define the path of the directory that will store client request body files. An additional option lets you separate those files into a folder hierarchy over as many as three levels.

Syntax: `client_body_temp_path path [level1] [level2] [level3]`

Default value: `client_body_temp`

Examples:

```
client_body_temp_path /tmp/nginx_rbf;
client_body_temp_path temp 2; # Nginx will create 2-digit folders to
hold request body files
client_body_temp_path temp 1 2 4; # Nginx will create 3 levels of
folders (first level: 1 digit, second level: 2 digits, third level: 4
digits)
```

client_body_timeout

Context: `http`, `server`, and `location`

Defines the inactivity timeout while reading a client request body. A connection becomes inactive the moment the client stops transmitting data. If the delay is reached, NGINX returns a `408 Request timeout` HTTP error.

Syntax: Time value (in seconds)

Default value: `60`

client_header_buffer_size

Context: `http`, `server`, and `location`

This directive allows you to define the size of the buffer that NGINX allocates to request headers. Usually, `1k` is enough. However, in some cases, headers contain large chunks of cookie data, or the request URI is lengthy. If that is the case, then NGINX allocates one or more larger buffers (the size of larger buffers is defined by the `large_client_header_buffers` directive).

Syntax: Size value

Default value: `1k`

client_header_timeout

Context: `http`, `server`, and `location`

Defines the inactivity timeout while reading a client request header. A connection becomes inactive the moment the client stops transmitting data. If the delay is reached, NGINX returns a `408 Request timeout` HTTP error.

Syntax: Time value (in seconds)

Default value: `60`

client_max_body_size

Context: `http`, `server`, and `location`

This is the maximum size of a client request body. If this size is exceeded, NGINX returns a `413 Request entity too large` HTTP error. This setting is particularly important if you are going to allow users to upload files to your server over HTTP.

Syntax: Size value

Default value: `1m`

large_client_header_buffers

Context: `http`, `server`, and `location`

Defines the number and size of larger buffers to be used for storing client requests, in the event the default buffer (`client_header_buffer_size`) was insufficient. Each line of the header must fit in the size of a single buffer. If the request URI line is greater than the size of a single buffer, NGINX returns a `414 Request URI too large` error. If another header line exceeds the size of a single buffer, NGINX returns a `400 Bad request` error.

Syntax: `large_client_header_buffers amount size`

Default value: *4*8* kilobytes

lingering_time

Context: `http`, `server`, and `location`

This directive applies to client requests with a request body. As soon as the amount of uploaded data exceeds `max_client_body_size`, NGINX immediately sends a `413 Request entity too large` HTTP error response. However, most browsers continue uploading data regardless of that notification. This directive defines the amount of time NGINX should wait after sending this error response before closing the connection.

Syntax: Numeric value (time)

Default value: `30` seconds

lingering_timeout

Context: `http`, `server`, and `location`

This directive defines the amount of time that NGINX should wait between two read operations before closing the client connection.

Syntax: Numeric value (time)

Default value: `5` seconds

lingering_close

Context: `http`, `server`, and `location`

Controls the way NGINX closes client connections. Set this to `off` to immediately close connections after all request data has been received. The default value (`on`) allows waiting time to process additional data if necessary. If set to `always`, NGINX will always wait to close the connection. The amount of waiting time is defined by the `lingering_timeout` directive.

Syntax: `on`, `off`, or `always`

Default value: `on`

ignore_invalid_headers

Context: `http` and `server`

If this directive is disabled, NGINX returns a `400 Bad Request` HTTP error if request headers are malformed.

Syntax: `on` or `off`

Default value: `on`

chunked_transfer_encoding

Context: `http`, `server`, and `location`

Enables or disables chunked transfer encoding for HTTP/1.1 requests.

Syntax: `on` or `off`

Default value: `on`

max_ranges

Context: `http`, `server`, and `location`

Defines how many byte ranges NGINX will serve when a client requests partial content from a file. If you do not specify a value, there is no limit. If you set this to `0`, the byte range functionality is disabled.

Syntax: Size value

MIME types

NGINX offers two particular directives that will help you configure MIME types: `types` and `default_type`, which defines the default MIME types for documents. This will affect the `Content-Type` HTTP header sent within responses. Read on.

types

Context: `http`, `server`, and `location`

This directive allows you to establish correlations between MIME types and file extensions. It's actually a block accepting a particular syntax:

```
types {
  mimetype1  extension1;
  mimetype2  extension2 [extension3...];
  [...]
}
```

When NGINX serves a file, it checks the file extension in order to determine the MIME type. The MIME type is then sent as the value of the `Content-Type` HTTP header in the response. This header may affect the way browsers handle files. For example, if the MIME type of the file you are requesting is `application/pdf`, your browser may, for instance, attempt to render the file using a plugin associated with that MIME type instead of merely downloading it.

NGINX includes a basic set of MIME types as a standalone file (`mime.types`) to be included with the `include` directive:

```
include mime.types;
```

This file already covers the most important file extensions, so you will probably not need to edit it. If the extension of the served file is not found within the listed types, the default type is used, as defined by the `default_type` directive (see the next entry).

Note that you may override the list of types by re-declaring the `types` block. A useful example would be to force all files in a folder to be downloaded instead of being displayed:

```
http {
    include mime.types;
    [...]
    location /downloads/ {
        # removes all MIME types
        types { }
        default_type application/octet-stream;
     }
   [...]
}
```

Some browsers ignore MIME types and may still display files if their filename ends with a known extension, such as `.html` or `.txt`.

To control the way files are handled by your visitors' browsers in a more certain and definitive manner, you should make use of the `Content-Disposition` HTTP header via the `add_header` directive, detailed in the *HTTP headers* module (*Chapter 4*).

The default values, if the `mime.types` file is not included, are as follows:

```
types {
  text/html html;
  image/gif gif;
  image/jpeg jpg;
}
```

default_type

Context: `http`, `server`, and `location`

Defines the default MIME type. When NGINX serves a file, the file extension is matched against the known types declared within the `types` block in order to return the proper MIME type as a value of the `Content-Type` HTTP response header. If the extension doesn't match any of the known MIME types, the value of the `default_type` directive is used.

Syntax: MIME type

Default value: `text/plain`

types_hash_max_size

Context: `http`, `server`, and `location`

Defines the maximum size of an entry in the MIME types' hash tables.

Syntax: Numeric value

Default value: `4k` or `8k` (one line of CPU cache)

types_hash_bucket_size

Context: `http`, `server`, and `location`

Sets the bucket size for the MIME types' hash tables. You should only change this value if NGINX tells you to.

Syntax: Numeric value

Default value: `64`

Limits and restrictions

This set of directives will allow you to add restrictions that apply when a client attempts to access a particular location or document on your server. Note that you will find additional directives for restricting access in the next chapter.

limit_except

Context: `location`

This directive allows you to prevent the use of all HTTP methods, except ones that you explicitly allow. Within a `location` block, you may want to restrict the use of some HTTP methods, such as forbidding clients from sending `POST` requests. You need to define two elements: first, methods that are not forbidden (allowed methods; all others will be forbidden); and second, the audience that is affected by the restriction:

```
location /admin/ {
    limit_except GET {
      allow 192.168.1.0/24;
      deny all;
    }
}
```

This example applies a restriction to the `/admin/` location—all visitors are only allowed to use the `GET` method. Visitors that have a local IP address, as specified with the `allow` directive (detailed in the HTTP access module), are not affected by this restriction. If a visitor uses a forbidden method,

NGINX will return a `403 Forbidden` HTTP error. Note that the `GET` method implies the `HEAD` method (if you allow `GET`, both `GET` and `HEAD` are allowed).

The syntax is specific:

```
limit_except METHOD1 [METHOD2...] {
  allow | deny | auth_basic | auth_basic_user_file | proxy_pass |
perl;
}
```

Directives that you are allowed to insert within the block are documented in their respective module section in *Chapter 4*.

limit_rate

Context: `http`, `server`, `location`, and `if`

Allows you to limit the transfer rate of individual client connections. The rate is expressed in bytes per second:

```
limit_rate 500k;
```

This will limit connection transfer rates to `500` kilobytes per second. If a client opens two connections, the client will be allowed *2*500* kilobytes.

Syntax: Size value

Default value: No limit

limit_rate_after

Context: `http`, `server`, `location`, and `if`

Defines the number of data transferred before the `limit_rate` directive takes effect:

```
limit_rate 10m;
```

NGINX will send the first 10 megabytes at maximum speed. Past this size, the transfer rate is limited by the value specified with the `limit_rate` directive (see the previous entry). Similar to the `limit_rate` directive, this setting only applies to a single connection.

Syntax: Size value

Default: None

satisfy

Context: `location`

The `satisfy` directive defines whether clients require all access conditions to be valid (satisfy all), or at least one (satisfy any):

```
location /admin/ {
    allow 192.168.1.0/24;
    deny all;
    auth_basic "Authentication required";
    auth_basic_user_file conf/htpasswd;
}
```

In the previous example, there are two conditions for clients to be able to access the resource:

- Through the `allow` and `deny` directives (HTTP access module), we only allow clients that have a local IP address; all other clients are denied access
- Through the `auth_basic` and `auth_basic_user_file` directives (the HTTP `auth_basic` module), we only allow clients that provide a valid username and password

With `satisfy all`, the client must satisfy both conditions in order to gain access to the resource. With `satisfy any`, if the client satisfies either condition, they are granted access.

Syntax: `satisfy any | all`

Default value: `all`

internal

Context: `location`

This directive specifies that the `location` block is internal. In other words, the specified resource cannot be accessed by external requests:

```
server {
    [...]
    server_name .website.com;
    location /admin/ {
        internal;
    }
}
```

With the previous configuration, clients will not be able to browse `http://website.com/admin/`. Such requests will be met with `404 Not Found` errors. The only way to access the resource is via internal redirects.

File processing and caching

It's important for your websites to be built upon solid foundations. File access and caching are critical aspects of web serving. In this regard, NGINX lets you perform precise tweaking with the use of the following directives.

disable_symlinks

This directive allows you to control the way NGINX handles symbolic links when they are to be served. By default (the directive value is `off`), symbolic links are allowed, and NGINX follows them. You may decide to disable the following symbolic links under different conditions by specifying one of these values:

- `on`: If any part of the requested URI is a symbolic link, access to it is denied, and NGINX returns a `403 HTTP` error page.
- `if_not_owner`: Similar to the previous link, but access is denied only if the link and the object it points to have different owners.
- The optional `from=` parameter allows you to specify a part of the URL that will not be checked for symbolic links. For example, `disable_symlinks on from=$document_root` will tell NGINX to normally follow symbolic links in the URI up to the `$document_root` folder. If a symbolic link is found in the URI parts after that, access to the requested file will be denied.

directio

Context: `http`, `server`, and `location`

If this directive is enabled, files with a size greater than the specified value will be read with the Direct I/O system mechanism. This allows NGINX to read data from the storage device and place it directly in memory with no intermediary caching process involved.

Syntax: Size value, or `off`

Default value: `off`

directio_alignment

Context: `http`, `server`, and `location`

Sets byte alignment when using `directio`. Set this value to `4k` if you use XFS under Linux.

Syntax: Size value

Default value: `512`

open_file_cache

Context: `http`, `server`, and `location`

This directive allows you to enable the cache that stores information about open files. It does not actually store file contents but only information such as the following:

- File descriptors (file size, modification time, and so on).
- The existence of files and directories.
- File errors, such as permission denied, file not found, and so on. Note that this can be disabled with the `open_file_cache_errors` directive.

This directive accepts two arguments:

- `max=X`, where `X` is the number of entries that the cache can store. If this number is reached, older entries will be deleted in order to leave room for newer entries.
- Optionally, `inactive=Y`, where `Y` is the of seconds that a cache entry should be stored. By default, NGINX will wait *60* seconds before clearing a cache entry. If the cache entry is accessed, the timer is reset. If the cache entry is accessed more than the value defined by `open_file_cache_min_uses`, the cache entry will not be cleared (until NGINX runs out of space and decides to clear out older entries).

Syntax: `open_file_cache max=X [inactive=Y] | off`

Default value: `off`

Example:

```
open_file_cache max=5000 inactive=180;
```

open_file_cache_errors

Context: `http`, `server`, and `location`

Enables or disables the caching of file errors with the `open_file_cache` directive (read the previous entry).

Syntax: `on` or `off`

Default value: `off`

open_file_cache_min_uses

Context: `http`, `server`, and `location`

By default, entries in the `open_file_cache` directive are cleared after a period of inactivity (60 seconds, by default). If there is activity, though, you can prevent NGINX from removing the cache entry. This directive defines the number of times an entry must be accessed in order to be eligible for protection:

```
open_file_cache_min_uses 3;
```

If the cache entry is accessed more than three times, it becomes permanently active and is not removed until NGINX decides to clear out older entries to free up some space.

Syntax: Numeric value

Default value: `1`

open_file_cache_valid

Context: `http`, `server`, and `location`

The open file cache mechanism is important, but cached information quickly becomes obsolete, especially in the case of a fast-moving filesystem. In that regard, information needs to be re-verified after a short period of time. This directive specifies the number of seconds that NGINX will wait before revalidating a cache entry.

Syntax: Time value (in seconds)

Default value: `60`

read_ahead

Context: `http`, `server`, and `location`

Defines the number of bytes to pre-read from files. Under Linux-based operating systems, setting this directive to a value above `0` will enable reading ahead, but the actual value you specify has no effect. Set this to `0` to disable pre-reading.

Syntax: Size value

Default value: `0`

Other directives

The following directives relate to various aspects of web server logging, URI composition, DNS, and so on.

log_not_found

Context: `http`, `server`, and `location`

Enables or disables logging of `404 Not Found` HTTP errors. If your logs get filled with `404` errors due to missing `favicon.ico` or `robots.txt` files, you might want to turn this off.

Syntax: `on` or `off`

Default value: `on`

log_subrequest

Context: `http`, `server`, and `location`

Enables or disables logging of sub-requests triggered by internal redirects or **server-side includes** (**SSI**) requests.

Syntax: `on` or `off`

Default value: `off`

merge_slashes

Context: `http`, `server`, and `location`

Enabling this directive will have the effect of merging multiple consecutive slashes in a URI. It turns out to be particularly useful in situations resembling the following:

```
server {
    [...]
    server_name website.com;
    location /documents/ {
        type { }
        default_type text/plain;
   }
}
```

By default, if the client attempts to access `http://website.com//documents/` (note `//` in the middle of the URI), NGINX will return a `404 Not Found` HTTP error. If you enable this directive, the two slashes will be merged into one, and the location pattern will be matched.

Syntax: `on` or `off`

Default value: `off`

msie_padding

Context: `http`, `server`, and `location`

This directive functions with the **Microsoft Internet Explorer** (**MSIE**) and Google Chrome browser families. In the case of error pages (with error code `400` or higher), if the length of the response body is less than 512 bytes, these browsers will display their own error page, sometimes at the expense of a more informative page provided by the server. If you enable this option, the body of responses with a status code of `400` or higher will be padded to 512 bytes.

Syntax: `on` or `off`

Default value: `off`

msie_refresh

Context: `http`, `server`, and `location`

This is another MSIE-specific directive that will take effect in the case of the following HTTP response codes, `301 Moved Permanently` and `302 Moved Temporarily`. When enabled, NGINX sends clients running an MSIE browser a response body containing a refresh meta tag (`<meta http-equiv="Refresh"...>`) in order to redirect the browser to the new location of the requested resource.

Syntax: `on` or `off`

Default value: `off`

resolver

Context: `http`, `server`, and `location`

Specifies the name servers that should be employed by NGINX to resolve hostnames to IP addresses and vice versa. DNS query results are cached for some time, either by respecting the **time to live** (**TTL**) provided by the DNS server or by specifying a time value to the valid argument.

If more than one DNS server is specified, NGINX will query them using a round-robin algorithm.

Syntax: One or more IPv4 or IPv6 addresses, `valid=Time value, ipv6=on|off`

Default value: None (system default)

Examples:

```
resolver 127.0.0.1; #use local DNS
resolver 8.8.8.8 8.8.4.4 valid=1h;#GoogleDNS & 1 hour cache
```

> **Important note**
>
> Although NGINX supports external DNS resolvers, we strongly recommend you use a local resolver such as `dnsmasq` and have NGINX query `dnsmasq` instead. NGINX might struggle to work with external resolvers under heavy load, and having a local resolver removes any network issue such as latency between NGINX and the DNS server. By doing so, you can optimize `dnsmasq` or use an alternative such as `dnscrypt` to keep your DNS requests private. Once set, a local DNS server can be used for any other server application.

resolver_timeout

Context: `http`, `server`, and `location`

Timeout for a hostname resolution query.

Syntax: Time value (in seconds)

Default value: `30`

server_tokens

Context: `http`, `server`, and `location`

This directive allows you to define whether or not NGINX should inform clients of the running version number. There are three situations where NGINX indicates its version number:

- In the server header of HTTP responses (such as `nginx/1.25.0`). If you set `server_tokens` to `off`, the server header will only indicate `Nginx`.
- On error pages, NGINX indicates the version number in the footer. If you set `server_tokens` to `off`, the footer on error pages will only indicate `Nginx`.
- If using the build value, NGINX will output the build value specified during compilation.

If you are running an older version of NGINX and do not plan to update it, it might be a good idea to *hide* your version number for security reasons.

Syntax: `on`, `off`, or `build`. Default value: `on`.

underscores_in_headers

Context: `http` and `server`

Allows or disallows underscores in custom HTTP header names. If this directive is set to `on`, the following example header is considered valid by `Nginx: test_header: value`:

Syntax: `on` or `off`

Default value: `off`

variables_hash_max_size

Context: `http`

This directive defines the maximum size of variable hash tables. If your server configuration uses a total of more than 1,024 variables, you will have to increase this value.

Syntax: Numeric value

Default value: `1024`

variables_hash_bucket_size

Context: `http`

This directive allows you to set the bucket size for variable hash tables.

Syntax: Numeric value

Default value: `64` (or `32` or `128`, depending on your processor cache specifications)

post_action

Context: `http`, `server`, `location`, and `if`

Defines a post-completion action, a URI that will be called by NGINX after the request has been completed.

Syntax: URI or named `location` block

Example:

```
location /payment/ {
    post_action /scripts/done.php;
}
```

We covered most of the directives we might need in the future. Here's an example of another set of optional directives that we recommend you use. Please note that at the time of writing this book, HTTP/3 and **Quick UDP Internet Connections** (**QUIC**) are neither fully supported nor deployed in NGINX.

Exploring the directives of HTTP/2

NGINX added support for **HTTP/2** in version `1.9.5` and superseded the **SPDY** (pronounced **SPeeDY**) module, meaning that, as of `1.9.5`, SPDY is no longer available, and we now have to use HTTP/2.

If you installed NGINX via a package manager, this module is most likely enabled; if you compiled it yourself, please make sure you compiled NGINX using the `--with_http_v2_module` configure flag.

Similarly to SPDY, HTTP/2 requires the use of SSL, which is good practice regardless. These days, SSL certificates can be issued for free by services such as *Let's Encrypt*, so this is highly recommended.

To enable HTTP/v2, add the `http2` flag to the `listen` directive:

```
listen 443 ssl http2;
```

Let's explore the different module directives in this section.

http2_chunk_size

Context: `http`, `server`, and `location`

Sets the maximum size of chunks into which the response body is sliced.

Syntax: Size

Default value: `8k`

http2_body_preread_size

Context: `http` and `server`

Sets the size of the request buffer in which the body may be saved before it is processed.

Syntax: Size

Default value: `64k`

http2_idle_timeout

Context: `http` and `server`

Sets the time after which the connection is closed due to inactivity.

Syntax: Time

Default value: `3m`

http2_max_concurrent_streams

Context: `http` and `server`

Sets the maximum number of concurrent HTTP/2 streams in a connection.

Syntax: Number

Default value: `128`

http2_max_field_size

Context: `http` and `server`

Limits the maximum size of a compressed request header field.

Syntax: Size

Default value: `4k`

http2_max_header_size

Context: `http` and `server`

Limits the maximum size of the entire request header list after decompression.

Syntax: Size

Default value: `16k`

http2_max_requests

Context: `http` and `server`

Sets the maximum number of requests that can be served through one HTTP/2 connection, after which the connection is closed and the client should use a new connection.

Syntax: Number

Default value: `1000`

http2_recv_buffer_size

Context: `http`

Sets the size of the per-worker input buffer.

Syntax: Size

Default value: `256k`

http2_recv_timeout

Context: `http` and `server`

Sets the timeout for expecting more data from the client, after which the connection is closed.

Syntax: Time

Default value: `30`

Module variables

The HTTP/2 module sets only a single variable to determine whether HTTP/2 is used or not:

Variable	Description
`$http2`	`h2` if HTTP/2 over TLS, `h2c` if over cleartext TCP. Empty string (( )) if HTTP/2 is not used.

Now that we have finished covering most of the directives, let's move on to variables and the extended configuration of these directives.

Exploring the variables introduced by the HTTP core module

The HTTP core module introduces a large set of variables that you can use within the value of directives. Be careful, though, as only a handful of directives accept variables in the definition of their value. If you insert a variable in the value of a directive that does not accept variables, no error is reported; instead, the variable name appears as raw text.

There are three different kinds of variables that you will come across. The first set represents the values transmitted in the headers of the client request. The second set corresponds to the headers of the response sent to the client. Finally, the third set comprises variables that are completely generated by NGINX.

Request headers

NGINX lets you access client request headers in the form of variables that you will be able to employ later on in the configuration:

Variable	Description
`$http_host`	Value of the `Host` HTTP header, a string indicating the hostname that the client is trying to reach.
`$http_user_agent`	Value of the `User-Agent` HTTP header, a string indicating the web browser of the client.
`$http_referer`	Value of the `Referer` HTTP header, a string indicating the URL of the previous page from which the client comes.
`$http_via`	Value of the `Via` HTTP header, which informs us about possible proxies used by the client.
`$http_x_forwarded_for`	Value of the `X-Forwarded-For` HTTP header, which shows the actual IP address of the client if the client is behind a proxy.
`$http_cookie`	Value of the `Cookie` HTTP header, which contains cookie data sent by the client.

Variable	Description
`$http_...`	Additional headers sent by the client can be retrieved using `$http_` followed by the header name in lowercase and with dashes (`-`) replaced by underscores (`_`).

Table 3.1: A list of customizable HTTP request headers

Response headers

In a similar fashion, you are allowed to access the HTTP headers of the response that was sent to the client. These variables are not available at all times—they will only carry a value after the response is sent; for instance, at the time of writing messages in the logs:

Variable	Description
`$sent_http_content_type`	Value of the `Content-Type` HTTP header, indicating the MIME type of the resource being transmitted.
`$sent_http_content_length`	Value of the `Content-Length` HTTP header, informing the client of the response body length.
`$sent_http_location`	Value of the `Location` HTTP header, which indicates that the location of the desired resource is different from the one specified in the original request.
`$sent_http_last_modified`	Value of the `Last-Modified` HTTP header, corresponding to the modification date of the requested resource.
`$sent_http_connection`	Value of the `Connection` HTTP header, defining whether the connection will be kept alive or closed.
`$sent_http_keep_alive`	Value of the `Keep-Alive` HTTP header that defines the amount of time a connection will be kept alive.
`$sent_http_transfer_encoding`	Value of the `Transfer-Encoding` HTTP header, giving information about the response body encoding method (such as compress, gzip).
`$sent_http_cache_control`	Value of the `Cache-Control` HTTP header, telling us whether the client browser should cache the resource or not.
`$sent_http_...`	Additional headers sent to the client can be retrieved using `$sent_http_` followed by the header name, in lowercase and with dashes (`-`) replaced by underscores (`_`).

Table 3.2: A list of customizable HTTP response headers

NGINX-generated headers

Apart from HTTP headers, NGINX provides a large of variables concerning the request, the way it was and will be handled, as well as settings in use with the current configuration:

Variable	Description
`$arg_XXX`	Allows you to access the query string (`GET` parameters), where `XXX` is the name of the parameter you want to utilize.
`$args`	All of the arguments of the query string combined together.
`$binary_remote_addr`	IP address of the client as binary data (4 bytes).
`$body_bytes_sent`	Amount of bytes sent in the body of the response (does not include response headers).
`$bytes_sent`	Amount of bytes sent to the client.
`$connection`	Serial number identifying a connection.
`$connection_requests`	Amount of requests already served by the current connection.
`$content_length`	Equates to the `Content-Length` HTTP header.
`$content_type`	Equates to the `Content-Type` HTTP header.
`$cookie_XXX`	Allows you to access cookie data where `XXX` is the name of the parameter you want to utilize.
`$document_root`	Returns the value of the `root` directive for the current request.
`$document_uri`	Returns the current URI of the request. It may differ from the original request URI if internal redirects were performed. It is identical to the `$uri` variable.
`$host`	This variable equates to the `host` HTTP header of the request. NGINX itself gives this variable a value for cases where the `host` header is not provided in the original request.
`$hostname`	Returns the system hostname of the server computer.
`$https`	Set to `on` for HTTPS connections; empty otherwise.
`$is_args`	If the `$args` variable is defined, `$is_args` equates to `?`. If `$args` is empty, `$is_args` is empty as well. You may use this variable for constructing a URI that optionally comes with a query string, such as `index.php$is_args$args`. If there is any query string argument in the request, `$is_args` is set to `?`, making this a valid URI.
`$limit_rate`	Returns the per-connection transfer rate limit, as defined by the `limit_rate` directive. You are allowed to edit this variable by using `set` (directive from the Rewrite module): `set $limit_rate 128k;`.

Variable	Description
`$msec`	Returns the current time (in seconds and milliseconds).
`$nginx_version`	Returns the version of NGINX you are running.
`$pid`	Returns the NGINX process identifier.
`$pipe`	If the current request is pipelined, this variable is set to `p`; otherwise, the value is `[.]`.
`$proxy_protocol_addr`	If the `proxy_protocol` parameter is enabled on the `listen` directive, this variable will contain the client address.
`$query_string`	Identical to `$args`.
`$remote_addr`	Returns the IP address of the client.
`$remote_port`	Returns the port of the client socket.
`$remote_user`	Returns the client username if they used authentication.
`$realpath_root`	Returns the document root in the client request, with symbolic links resolved into the actual path.
`$request_body`	Returns the body of the client request, or – if the body is empty.
`$request_body_file`	If the request body was saved (see the `client_body_in_file_only` directive), this variable indicates the path of the temporary file.
`$request_completion`	Returns `OK` if the request is completed; an empty string otherwise.
`$request_filename`	Returns the full filename served in the current request.
`$request_length`	Returns the total length of the client request.
`$request_method`	Indicates the HTTP method used in the request, such as `GET` or `POST`.
`$request_time`	Returns the amount of time elapsed since the first byte was read from the client (seconds and milliseconds value).
`$request_id`	Unique request identifier generated from 16 random bytes, in hexadecimal.
`$request_uri`	Corresponds to the original URI of the request; remains unmodified all through the process (unlike `$document_uri/$uri`).
`$scheme`	Returns either `http` or `https`, depending on the request.
`$server_addr`	Returns the IP address of the server. Be careful, as each use of the variable requires a system call, which could potentially affect overall performance in the case of high-traffic setups.
`$server_name`	Indicates the value of the `server_name` directive that was used while processing the request.

Variable	Description
`$server_port`	Indicates the port of the server socket that received the request data.
`$server_protocol`	Returns the protocol and version, usually HTTP/1.0 or HTTP/1.1.
`$status`	Returns the response status code.
`$tcpinfo_rtt, $tcpinfo_rttvar, $tcpinfo_snd_cwnd, $tcpinfo_rcv_space`	If your operating system supports the `TCP_INFO` socket option, these variables will be populated with information on the current client TCP connection.
`$time_iso8601, $time_local`	Provides the current time respectively in ISO 8601 and local formats for use with the `access_log` directive.
`$uri`	Identical to `$document_uri`.

Table 3.3: A list of various customizable variables; these variables are optional but might come in handy

Understanding and exploring the location block

We have established that NGINX lets you fine-tune your configuration down to three levels—at the protocol level (the `http` block), the server level (the `server` block), and the requested URI level (the `location` block). Let's now go into more detail about the third one.

Location modifier

NGINX allows you to define `location` blocks by specifying a pattern that will be matched against the requested document URI:

```
server {
    server_name website.com;
    location /admin/ {
    # The configuration you place here only applies to
    # http://website.com/admin/
    }
}
```

Instead of a simple folder name, you can indeed insert complex patterns. The syntax of the `location` block is shown here:

```
location [=|~|~*|^~|@] pattern { … }
```

The first optional argument is a symbol called `location` that will define the way NGINX matches the specified pattern and also defines the very nature of the pattern (a simple string or regular expression). The underlying subsections detail the different modifiers and their behavior.

The = modifier

The requested document URI must match the specified pattern exactly. The pattern here is limited to a simple literal string; you cannot use a regular expression:

```
server {
    server_name website.com;
    location = /abcd {
    [...]
    }
}
```

The configuration in the `location` block has the following attributes:

- Applies to `http://website.com/abcd` (exact match)
- May apply to `http://website.com/ABCD` (it is case-sensitive if your operating system uses a case-sensitive filesystem)
- Applies to `http://website.com/abcd?param1&param2` (regardless of query string arguments)
- Does not apply to `http://website.com/abcd/` (trailing slash)
- Does not apply to `http://website.com/abcde` (extra characters after the specified pattern)

No modifier

The requested document URI must begin with the specified pattern. You may not use regular expressions:

```
server {
    server_name website.com;
    location /abcd {
    [...]
    }
}
```

The configuration in the `location` block has the following attributes:

- Applies to `http://website.com/abcd` (exact match)
- May apply to `http://website.com/ABCD` (it is case-sensitive if your operating system uses a case-sensitive filesystem)
- Applies to `http://website.com/abcd?param1&param2` (regardless of query string arguments)
- Applies to `http://website.com/abcd/` (trailing slash)
- Applies to `http://website.com/abcde` (extra characters after the specified pattern)

The ~ modifier

The requested URI must be a case-sensitive match for the specified regular expression:

```
server {
    server_name website.com;
    location ~ ^/abcd$ {
    [...]
    }
}
```

The `^/abcd$` regular expression used in this example specifies that the pattern must begin (`^`) with `/`, be followed by `abc`, and finish (`$`) with `d`. Consequently, the configuration in the `location` block has the following attributes:

- Applies to `http://website.com/abcd` (exact match)
- Does not apply to `http://website.com/ABCD` (case-sensitive)
- Applies to `http://website.com/abcd?param1&param2` (regardless of query string arguments)
- Does not apply to `http://website.com/abcd/` (trailing slash) due to the specified regular expression
- Does not apply to `http://website.com/abcde` (extra characters) due to the specified regular expression

> **Note**
> With operating systems such as Microsoft Windows, ~ and ~* are both case-insensitive, as the OS uses a case-insensitive filesystem.

The ~ modifier*

The requested URI must be a case-insensitive match for the specified regular expression:

```
server {
    server_name website.com;
    location ~* ^/abcd$ {
    [...]
    }
}
```

The regular expression used in the example is similar to the previous one. Consequently, the configuration in the `location` block has the following attributes:

- Applies to `http://website.com/abcd` (exact match)
- Applies to `http://website.com/ABCD` (case-insensitive)
- Applies to `http://website.com/abcd?param1&param2` (regardless of query string arguments)
- Does not apply to `http://website.com/abcd/` (trailing slash) due to the specified regular expression
- Does not apply to `http://website.com/abcde` (extra characters) due to the specified regular expression

The ^~ modifier

Similar to the no-symbol behavior, the location URI must begin with the specified pattern. The difference is that, if the pattern is matched, NGINX stops searching for other patterns (read the *Search order and priority* section).

The @ modifier

Defines a named `location` block. These blocks cannot be accessed by the client but only by internal requests generated by other directives, such as `try_files` or `error_page`.

Search order and priority

Since it's possible to define multiple `location` blocks with different patterns, you need to understand that when NGINX receives a request, it searches for the `location` block that best matches the requested URI:

```
server {
    server_name website.com;
    location /files/ {
    # applies to any request starting wi"h "/fil"s/"
    # for example /files/doc.txt, /files/, /files/temp/
    }
    location = /files/ {
    # applies to the exact request "o "/fil"s/"
    # and as such does not apply to /files/doc.txt
    # but only /files/
    }
}
```

When a client visits `http://website.com/files/doc.txt`, the first `location` block applies. However, when they visit `http://website.com/files/`, the second block applies (even though the first one matches) because it has priority over the first one (it is an exact match).

The order you established in the configuration file (placing the `/files/` block before the `= /files/` block) is irrelevant. NGINX will search for matching patterns in a specific order:

- `location` blocks with the = modifier: If the specified string exactly matches the requested URI, NGINX retains the `location` block
- `location` blocks with no modifier: If the specified string *exactly* matches the requested URI, NGINX retains the `location` block
- `location` blocks with the `^~` modifier: If the specified string matches the beginning of the requested URI, NGINX retains the `location` block
- `location` blocks with the ~ or `~*` modifier: If the regular expression matches the requested URI, NGINX retains the `location` block
- `location` blocks with no modifier: If the specified string matches the *beginning* of the requested URI, NGINX retains the `location` block

To that extent, the `^~` modifier begins to make sense, and we can envision cases where it becomes useful, as shown in the following three subsections.

Case 1

Here's an example of two `location` blocks both starting with `/doc`:

```
server {
    server_name website.com;
    location /doc {
    [...] # requests beginning with "/doc"
    }
    location ~* ^/document$ {
    [...] # requests exactly matching "/document"
    }
}
```

You might wonder: When a client requests `http://website.com/document`, which of these two `location` blocks applies? Indeed, both blocks match this request. Again, the answer does not lie in the order in which the blocks appear in the configuration files. In this case, the second `location` block will apply as the `~*` modifier has priority over the other.

Case 2

Here's an example of two `location` blocks for a similar URL. The first `location` block starts with `/document[+]`, while the last one is an exact match for `/document`:

```
server {
    server_name website.com;
    location /document {
    [...] # requests beginning with "/document"
    }
    location ~* ^/document$ {
    [...] # requests exactly matching "/document"
    }
}
```

The question remains the same: What happens when a client sends a request to download `http://website.com/document`? There is a trick here. The string specified in the first block now exactly matches the requested URI. As a result, NGINX prefers it over the regular expression.

Case 3

Here's an example of a corresponding string to the second `location` block and being processed by the first `location` block:

```
server {
    server_name website.com;
    location ^~ /doc {
    [...] # requests beginning with "/doc"
    }
    location ~* ^/document$ {
    [...] # requests exactly matching "/document"
    }
}
```

This last case makes use of the `^~` modifier. Which block applies when a client visits `http://website.com/document`? The answer is the first block. The reason is that `^~` has priority over `~*`. As a result, any request with a URI beginning with `/doc` will be affected by the first block, even if the request URI matches the regular expression defined in the second block.

We've now covered the `location` block, its modifiers, and reusable use cases as we learned to prioritize certain requests according to their paths.

Summary

All through this chapter, we studied key concepts of the NGINX HTTP configuration. First, we learned about creating virtual hosts by declaring `server` blocks. Then, we discovered the directives and variables of the HTTP core module that can be inserted within those blocks and eventually understood the mechanisms governing the `location` block.

The job is done. Your server now actually serves websites. We are going to take it one step further by discovering the modules that truly form the powerhouse of NGINX. The next chapter will deal with advanced topics, such as the rewrite and SSI modules, as well as additional components of the HTTP server.

4

Exploring Module Configuration in NGINX

The true power of NGINX lies within its modules. The entire application is built on a modular system, and each module can be enabled or disabled at compile time. Some bring up simple functionalities, such as the autoindex module, which generates a listing of the files in a directory. Some will transform your perception of a web server (such as the Rewrite module). Apart from the NGINX modules that already exist, developers can also create their own modules to suit their needs.

A quick overview of the third-party module system can be found at the end of this chapter. Please note that third-party modules are maintained by the community, and there is no guarantee that these modules will be compatible with your version of NGINX.

This chapter covers the following topics:

- The Rewrite module, which does more than just rewrite URIs
- Additional modules enabled in the default NGINX build
- Optional modules that must be enabled at compile time
- A quick note on third-party modules

Exploring the Rewrite module

This module, in particular, brings much more functionality to NGINX than a simple set of directives. It defines a whole new level of request processing that will be explained throughout this section.

Initially, the purpose of this module (as the name suggests) is to perform URL rewriting. This mechanism allows you to get rid of ugly URLs containing multiple parameters—for instance, `http://example.com/article.php?id=1234&comment=32`—such URLs being particularly uninformative and meaningless for a regular visitor.

Instead, links to your website will contain useful information that indicates the nature of the page you are about to visit. The URL given in the example becomes `http://website.com/article-1234-32-US-economy-strengthens.html`. This solution is not only more interesting for your visitors but also for search engines. URL rewriting is a key element of **search engine optimization** (**SEO**).

The principle behind this mechanism is simple: it consists of rewriting the URI of the client request after it is received, before serving the file. Once rewritten, the URI is matched against `location` blocks in order to find the configuration that should be applied to the request. The technique is further detailed in the coming sections.

Reminder of regular expressions

First and foremost, this module requires a certain understanding of regular expressions, also known as **regexes** or **regexps**. Indeed, URL rewriting is performed by the `rewrite` directive, which accepts a pattern followed by the replacement URI.

It is a vast topic; entire books are dedicated to explaining the ins and outs. However, the simplified approach that we are about to examine should be more than sufficient to make the most of the mechanism.

Purpose

The first question we must answer is: What is the purpose of regular expressions? To put it simply, the main purpose is to verify that a string of characters matches a given pattern. The pattern is written in a particular language that allows for defining extremely complex and accurate rules:

String	Pattern	Does it match?	Explanation
`hello`	`^hello$`	Yes	The string begins with the `h` character (`^h`), followed by `e`, `l`, `l`, and then finishes with `o` (`o$`).
`hell`	`^hello$`	No	The string begins with the `h` character (`^h`), followed by `e`, `l`, `l`, but does not finish with `o`.
`Hello`	`^hello$`	Depends	If the engine performing the match is case-sensitive, the string doesn't match the pattern.

Table 4.1: A list of patterns with explanations

This concept becomes a lot more interesting when complex patterns are employed, such as one that validates an email address: `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$`. Validating a well-formed email address programmatically would require a great deal of code, while all of the work can be done with a single regular expression pattern matching.

PCRE syntax

The syntax that NGINX employs originates from the **Perl Compatible Regular Expressions** (**PCRE**) library, which (if you remember *Chapter 2*) is a prerequisite for making your own build unless you disable modules that make use of it. It's the most commonly used form of regular expression, and nearly everything you learn here remains valid for other language variations.

In its simplest form, a pattern is composed of one character—for example, `x`. We can match strings against this pattern. Does `example` match the `x` pattern? Yes, `example` contains the `x` character. It can be more than one specific character; the `[a-z]` pattern matches any character between `a` and `z`, or even a combination of letters and digits: `[a-z0-9]`. In consequence, the `hell[a-z0-9]` pattern validates the following strings: `hello` and `hell4`, but not `hell` or `hell!`.

You probably noticed that we employed the `[` and `]` characters. These are called **metacharacters** and have a special effect on the pattern. There is a total of *11* metacharacters, and all play a different role. If you want to create a pattern that actually contains one of these characters, you need to escape the character with \ (backslash):

Metacharacter	Description
`^` Beginning	The entity after this character must be found at the beginning: • Example pattern: `^h` • Matching strings: `hello`, `h`, `hh` (anything beginning with *h*) • Non-matching strings: `character`, `ssh`
`$` End	The entity before this character must be found at the end: • Example pattern: `e$` • Matching strings: `sample`, `e`, `file` (anything ending with *e*) • Non-matching strings: `extra`, `shell`
`.` (dot) Any	Matches any character: • Example pattern: `hell.` • Matching strings: `hello`, `hellx`, `hell5`, and `hell!` • Non-matching strings: `hell`, `helo`

Metacharacter	Description
`[ ]` Set	Matches any character within the specified set: • Syntax: `[a-z]` for a range, `[abcd]` for a set, and `[a-z0-9]` for two ranges. Note that if you want to include the `-` character in a range, you need to insert it right after `[` or just before `]`. • Example pattern: `hell[a-y123-]` • Matching strings: `hello`, `hell1`, `hell2`, `hell3`, and `hell-` • Non-matching strings: `hellz`, `hell4`, `heloo`, and `he-llo`
`[^ ]` Negate set	Matches any character that is not within the specified set: • Example pattern: `hell[^a-np-z0-9]` • Matching strings: `hello`, and `hell!` • Non-matching strings: `hella`, `hell5`
`|` Alternation	Matches the entity placed either before or after `|`: • Example pattern: `hello|welcome` • Matching strings: `hello`, `welcome`, `helloes`, and `awelcome` • Non-matching strings: `hell`, `ellow`, `owelcom`
`( )` Grouping	Groups a set of entities, often to be used in conjunction with `|`. Also **captures** the matched entities; captures are detailed further on: • Example pattern: `^(hello|hi) there$` • Matching strings: `hello there`, `hi there` • Non-matching strings: `hey there`, `ahoy there`
`\` Escape	Allows you to escape special characters: Example pattern: `Hello\` Matching strings: `Hello.`, `Hello. How are you?`, `Hi! Hello...` Non-matching strings: `Hello`, `Hello!how are you?`

Table 4.2: A list of metacharacters and their roles

Quantifiers

So far, you are able to express simple patterns with a limited number of characters. Quantifiers allow you to extend the number of accepted entities:

Quantifier	Description
`*` 0 or more times	The entity preceding `*` must be found 0 or more times: • Example pattern: `he*llo` • Matching strings: `hllo`, `hello`, `heeeello` • Non-matching strings: `hallo`, `ello`
`+` 1 or more times	The entity preceding `+` must be found 1 or more times: • Example pattern: `he+llo` • Matching strings: `hello`, `heeeello` • Non-matching strings: `hllo`, `helo`
`?` 0 or 1 times	The entity preceding `?` must be found 0 or 1 times: • Example pattern: `he?llo` • Matching strings: `hello`, `hllo` • Non-matching strings: `heello`, `heeeello`
`{x}` `x` times	The entity preceding `{x}` must be found `x` times: • Example pattern: `he{3}llo` • Matching strings: `heeello`, `oh heeello there!` • Non-matching strings: `hello`, `heello`, `heeeello`
`{x,}` At least `x` times	The entity preceding `{x,}` must be found at least `x` times: • Example pattern: `he{3,}llo` • Matching strings: `heeello`, `heeeeeeello` • Non-matching strings: `hllo`, `hello`, `hello`
`{x,y}` `x` to `y` times	The entity preceding `{x,y}` must be found between `x` and `y` times: • Example pattern: `he{2,4}llo` • Matching strings: `heello`, `heeello`, `heeeello` • Non-matching strings: `hello`, `heeeeello`

Table 4.3: A list of quantifiers and their roles

As you probably noticed, the { and } characters in the regular expressions conflict with the block delimiter of the NGINX configuration file syntax language. If you want to write a regular expression pattern that includes *curly brackets*, you need to place the pattern between quotes (single or double quotes):

```
rewrite hel{2,}o /hello.php; # invalid
rewrite "hel{2,}o" /hello.php; # valid
rewrite 'hel{2,}o' /hello.php; # valid
```

Captures

One last feature of the regular expression mechanism is the ability to capture sub-expressions. Whatever text is placed between parentheses (( )) is captured and can be used after the matching process. The captured characters become available under the form of variables called `$N`, where `N` is a numeric index, in order of capture. Alternatively, you can attribute an arbitrary name to each of your captures (see the following example). Variables generated through the captures can be inserted within directive values.

Here are a couple of examples to illustrate the principle:

Pattern	**Example of a matching string**	**Captured**
`^(hello|hi) (sir|mister)$`	`hello sir`	`$1 = hello` `$2 = sir`
`^(hello (sir))$`	`hello sir`	`$1 = hello sir` `$2 = sir`
`^(.*)$`	`nginx rocks`	`$1 = nginx rocks`

`^(.{1,3})([0-9]{1,4})([?!]{1,2})$`	`abc1234!?`	`$1 = abc`
		`$2 = 1234`
		`$3 = !?`

Named captures are also supported through the following syntax: `?<name>` Example: `^/(?<folder>[^/]+)/(?<file>.*)$`	`/admin/doc`	`$folder = admin` `$file = doc`

Table 4.4: A list of patterns and expressions

When you use a regular expression in NGINX—for example, in the context of a `location` block—the buffers that you capture can be employed in later directives:

```
server {
    server_name website.com;
    location ~* ^/(downloads|files)/(.*)$ {
        add_header Capture1 $1;
        add_header Capture2 $2;
    }
}
```

In the preceding example, the `location` block will match the request URI against a regular expression. A few URIs that would apply here are `/downloads/file.txt`, `/files/archive.zip`, or even `/files/docs/report.doc`. Two parts are captured: `$1` will contain either `downloads` or `files`, and `$2` will contain whatever comes after `/downloads/` or `/files/`. Note that the `add_header` directive (syntax: `add_header header_name header_value`; see the *HTTP headers* section) is employed here to append arbitrary headers to the client response, for the sole purpose of demonstration.

Internal requests

NGINX differentiates external and internal requests. External requests originate directly from the client; the URI is then matched against possible `location` blocks:

```
server {
    server_name website.com;
    location = /document.html {
        deny all; # example directive
    }
}
```

A client request to `http://website.com/document.html` would directly fall into the preceding `location` block.

Opposite to this, internal requests are triggered by NGINX via specific directives. Among the directives offered by the default NGINX modules, there are several directives capable of producing internal requests: `error_page`, `index`, `rewrite`, `try_files`, `add_before_body`, and `add_after_body` (from the addition module), the `include` **Server-Side Includes** (**SSI**) command, and more.

There are two different types of internal requests:

- **Internal redirects**: NGINX redirects the client requests internally. The URI is changed, and the request may therefore match another `location` block and become eligible for different settings. The most common case of internal redirects is when using the `rewrite` directive, which allows you to rewrite the request URI.
- **Sub-requests**: Additional requests that are triggered internally to generate content that is complementary to the main request. A simple example would be with the addition module. The `add_after_body` directive allows you to specify a URI that will be processed after the original one, the resulting content being appended to the body of the original request. The SSI module also makes use of sub-requests to insert content with the `include` SSI command.

error_page

Detailed in the module directives of the NGINX HTTP core module, `error_page` allows you to define the server behavior when a specific error code occurs. The simplest form is to affect a URI to an error code:

```
server {
    server_name website.com;
    error_page 403 /errors/forbidden.html;
    error_page 404 /errors/not_found.html;
}
```

When a client attempts to access a URI that triggers one of these errors (such as loading a document or file that does not exist on the server, resulting in a `404` error), NGINX is supposed to serve the page associated with the error code. In fact, it does not just send the client the error page; it actually initiates a completely new request, based on the new URI.

Consequently, you can end up falling back on a different configuration, as in the following example:

```
server {
    server_name website.com;
    root /var/www/vhosts/website.com/httpdocs/;
    error_page 404 /errors/404.html;
    location /errors/ {
        alias /var/www/common/errors/;
        internal;
    }
}
```

When a client attempts to load a document that does not exist, they will initially receive a `404` error. We employed the `error_page` directive to specify that `404` errors should create an internal redirect to `/errors/404.html`. As a result, a new request is generated by NGINX, with the `/errors/404.html` URI. This URI falls under the location `/errors/` block, so the corresponding configuration applies.

> **Note**
>
> Logs can prove to be particularly useful when working with redirects and URL rewrites. Be aware that information on internal redirects will show up in the logs only if you set the `error_log` directive to `debug`. You can also get it to show up at the `notice` level, under the condition that you specify `rewrite_log on;` wherever you need it.

A raw but trimmed excerpt from the debug log summarizes the mechanism:

```
->http request line: "GET /page.html HTTP/1.1"
->http uri: "/page.html"
->test location: "/errors/"
->using configuration ""
->http filename: "/var/www/vhosts/website.com/httpdocs/page.html"
-> open() "/var/www/vhosts/website.com/httpdocs/page.html" failed (2:
No such file or directory), client: 127.0.0.1, server: website.com,
request: "GET /page.html HTTP/1.1", host:"website.com"
->http finalize request: 404, "/page.html?" 1
->http special response: 404, "/page.html?"
->internal redirect: "/errors/404.html?"
->test location: "/errors/"
->using configuration "/errors/"
->http filename: "/var/www/common/errors/404.html"
->http finalize request: 0, "/errors/404.html?" 1
```

Note that the use of the `internal` directive in the `location` block forbids clients from accessing the `/errors/` directory. This location can thus only be accessed through an internal redirect.

The mechanism is the same for the `index` directive (detailed further on in the *Index* section). If no file path is provided in the client request, NGINX will attempt to serve the specified index page by triggering an internal redirect.

Rewrite

While the previous `error_page` directive is not actually part of the Rewrite module, detailing its functionality provides a solid introduction to the way NGINX handles client requests.

Similar to how the `error_page` directive redirects to another location, rewriting the URI with the `rewrite` directive generates an internal redirect:

```
server {
    server_name website.com;
    root /var/www/vhosts/website.com/httpdocs/;
    location /storage/ {
        internal;
        alias /var/www/storage/;
    }
    location /documents/ {
        rewrite ^/documents/(.*)$ /storage/$1;
    }
}
```

A client query to `http://website.com/documents/file.txt` initially matches the second `location` block (`location /documents/`). However, the block contains a rewrite instruction that transforms the URI from `/documents/file.txt` to `/storage/file.txt`. The URI transformation reinitializes the process; the new URI is matched against the `location` blocks. This time, the first `location` block (`location /storage/`) matches the URI (`/storage/file.txt`).

Again, a quick peek at the debug log details the mechanism:

```
->http request line: "GET /documents/file.txt HTTP/1.1"
->http uri: "/documents/file.txt"
->test location: "/storage/"
->test location: "/documents/"
->using configuration "/documents/"
->http script regex: "^/documents/(.*)$"
->"^/documents/(.*)$" matches "/documents/file.txt", client:
127.0.0.1, server: website.com, request: "GET /documents/file.txt
HTTP/1.1", host: "website.com"
->rewritten data: "/storage/file.txt", args: "", client: 127.0.0.1,
server: website.com, request: "GET /documents/file.txt HTTP/1.1",
host: "website.com"
->test location: "/storage/"
->using configuration "/storage/"
->http filename: "/var/www/storage/file.txt"
->HTTP/1.1 200 OK
->http output filter "/storage/test.txt?"
```

Infinite loops

With all of the different syntaxes and directives, you could easily get confused. Worse, you might get NGINX confused. This happens, for instance, when your rewrite rules are redundant and cause internal redirects to *loop infinitely*:

```
server {
    server_name website.com;
    location /documents/ {
        rewrite ^(.*)$ /documents/2018/$1;
    }
}
```

You thought you were doing well, but this configuration actually triggers an internal redirect of `/documents/anything` to `/documents/2018//documents/anything`. Moreover, since the location patterns are re-evaluated after an internal redirect, `/documents/2018//documents/anything` becomes `/documents/2018//documents/2018//documents/anything`.

Here is the corresponding excerpt from the debug log:

```
->test location: "/documents/"
->using configuration "/documents/"
->rewritten data: "/documents/2018//documents/file.txt", [...]
->test location: "/documents/"
->using configuration "/documents/"
->rewritten data: "/documents/2018//documents/2018//documents/file.
txt" [...]
->test location: «/documents/»
->using configuration «/documents/»
->rewritten data: -
>»/documents/2018//documents/2018//documents/2018//documents/file.txt»
[...]
->[...]
```

You are probably wondering if this goes on indefinitely; the answer is *no*. The amount of cycles is restricted to *10*. You are only allowed 10 internal redirects. For anything past this limit, NGINX will produce a `500 Internal Server Error` message.

It is possible to prevent infinite loops for cases such as this where you might want to update the internal path over time. Simply use the `break` flag on the rewrite, as documented later, and NGINX will not do an internal redirect.

Conditional structure

The Rewrite module introduces a new set of directives and blocks, among which is the `if` conditional structure:

```
server {
    if ($request_method = POST) {
      [...]
    }
}
```

This offers the possibility to apply a configuration according to the specified condition. If the condition is `true`, the configuration is applied; otherwise, it isn't.

The following table describes the various syntaxes accepted when forming a condition:

Operator	Description

None	The condition is `true` if the specified variable or data is not equal to an empty string or a string starting with the character `0`:
	`if ($string) {`
	`  [...]`
	`}`

`=`, `!=`	The condition is `true` if the argument preceding the `=` symbol is equal to the argument following it. The following example can be read as, "If the request method is equal to `POST`, then apply the configuration":
	`if  ($request_method = POST) {`
	`  [...]`
	`}`
	The `!=` operator does the opposite: "If the request method is not equal to `GET`, then apply the configuration":
	`if  ($request_method != GET) {`
	`  [...]`
	`}`

Operator	Description

`~`, `~*`, `!~`, `!~*`	The condition is `true` if the argument preceding the ~ symbol matches the regular expression pattern placed after it:
	`if ($request_filename ~ "\.txt$") {`
	`  [...]`
	`}`
	~ is case-sensitive, and `~*` is case-insensitive. Use the `!` symbol to negate the matching:
	`if ($request_filename !~* "\.php$") {`
	`  [...]`
	`}`
	You can insert capture buffers in the regular expression:
	`if ($uri ~ "^/search/(.*)$") {`
	`  set $query $1;`
	`  rewrite ^ http://google.com/search?q=$query;`
	`}`

`-f`, `!-f`	Tests the existence of the specified file:
	`if (-f $request_filename) {`
	`  [...] # if the file exists`
	`}`
	Use `!-f` to test the non-existence of the file:
	`if (!-f $request_filename) {`
	`  [...] # if the file does not exist`
	`}`

`-d`, `!-d`	Similar to the `-f` operator, for testing the existence of a directory.
`-e`, `!-e`	Similar to the `-f` operator, for testing the existence of a file, directory, or symbolic link.
`-x`, `!-x`	Similar to the `-f` operator, for testing whether a file exists and is executable.

Table 4.5: A list of syntaxes for conditions

As of version 1.13.8, there are no `else` or `else if`-like instructions. However, other directives allowing you to control the configuration flow sequencing are available.

You might wonder: *What are the advantages of using a* `location` *block over an* `if` *block?* Indeed, in the following example, both seem to have the same effect:

```
if ($uri ~ /search/) {
  [...]
}
location ~ /search/ {
    [...]
}
```

As a matter of fact, the main difference lies within the directives that can be employed within either block. Some can be inserted in an `if` block and some can't; on the contrary, almost all directives are authorized within a `location` block, as you probably noticed in the directive listings so far. In general, it's best to only insert directives from the Rewrite module within an `if` block, as other directives were not originally intended for such usage.

Directives

The Rewrite module provides you with a set of directives that do more than just rewrite a URI. The following table describes these directives, along with the context in which they can be employed:

Directive	Description

`rewrite`
 	
`Context: server, location, if`
	As discussed previously, the `rewrite` directive allows you to rewrite the URI of the current request, thus resetting the treatment of the said request.
	
	Syntax: `rewrite regexp replacement [flag];`
	
	Where `regexp` is the regular expression, the URI should match in order for the replacement to apply.

Directive	Description
	The flag may take one of the following values: • `last`: The current rewrite rule should be the last to be applied. After its application, the new URI is processed by NGINX, and a `location` block is searched for. However, further rewrite instructions will be disregarded. • `break`: The current rewrite rule is applied, but NGINX does not initiate a new request for the modified URI (does not restart the search for matching `location` blocks). All further `rewrite` directives are ignored. • `redirect`: Returns a `302 Moved Temporarily` HTTP response, with the replacement URI set as the value of the `location` header.

	• `permanent`: Returns a `301 Moved Permanently` HTTP response, with the replacement URI set as the value of the `location` header.
	• If you specify a URI beginning with `http://` as the replacement URI, NGINX will automatically use the `redirect` flag.
	• Note that the request URI processed by the directive is a relative URI; it does not contain the hostname and protocol. For a request such as `http://website.com/documents/page.html`, the request URI is `/documents/page.html`.
	• If decoded, this URI corresponding to a request, such as `http://website.com/my%20page.html`, would be `/my page.html` (in the encoded URI, `%20` indicates a whitespace character).
	• Does not contain arguments; for a request such as `http://website.com/page.php?id=1&p=2`, the URI would be `/page.php`. When rewriting the URI, you don't need to consider including arguments in the replacement URI; NGINX does it for you. If you want NGINX to not include arguments after the rewritten URI, you must insert a `?` character at the end of the replacement URI: `rewrite ^/search/(.*)$ /search.php?q=$1?`.
	Examples:
	`rewrite ^/search/(.*)$ /search.php?q=$1;`
	`rewrite ^/search/(.*)$ /search.php?q=$1?;`
	`rewrite ^ http://website.com;`
	`rewrite ^ http://website.com permanent;`

Directive	Description

`break`
 	
Context: `server`, `location`, `if`
	The `break` directive is used to prevent further `rewrite` directives. Past this point, the URI is fixed and cannot be altered.
	
	Example:
	
	`if (-f $uri) {`
	`  break; # break  if the file exists`
	`}`
	`if ($uri ~  ^/search/(.*)$) {`
	`  set $query $1;`
	`  rewrite ^  /search.php?q=$query?;`
	`}`
	
	This example rewrites `/search/anything`-like queries to `/search.php?q=anything`. However, if the requested file exists (such as `/search/index.html`), the `break` instruction prevents NGINX from rewriting the URI.

`return`
 	
Context: `server`, `location`, `if`
	Interrupts the processing of the request and returns the specified HTTP status code or specified text.
	
	Syntax: `return code | text;`
	
	Here, the code is one of the following status codes: `204`, `308`, `400`, `402` to `406`, `408`, `410`, `411`, `413`, `416`, and `500` to `504`. In addition, you may use the `444` NGINX-specific code in order to return an `HTTP 200 OK` status code with no further response header or body. Alternatively, you may also specify a raw text value that will be returned to the user as a response body. This comes in handy when testing whether your request URIs fall within particular `location` blocks.
	
	Example:
	
	`if ($uri ~  ^/admin/) {`
	`  return 403;`
	`  # the  instruction below is NOT executed`
	`  # as Nginx  already completed the request`
	`  rewrite ^  http://website.com;`
	`}`

Directive	Description

`set`
 	
Context: `server`, `location`, `if`
	Initializes or redefines a variable. Note that some variables cannot be redefined; for example, you are not allowed to alter `$uri`.
	
	Syntax: `set $variable value;`
	
	Examples:
	
	`set $var1 "some   text";`
	
	`if ($var1 ~ ^(.*)   (.*)$) {`
	
	`  set $var2 $1$2;   #concatenation`
	
	`  rewrite ^   http://website.com/$var2;`
	
	`}`

`uninitialized_variable_warn` Context: `http`, `server`, `location`, `if`	If set to `on`, NGINX will issue log messages when the configuration employs a variable that has not yet been initialized. Syntax: `on` or `off` Example: `uninitialized_variable_warn on;`
`rewrite_log` Context: `http`, `server`, `location`, `if`	If `rewrite_log` is set to `on`, NGINX will issue log messages for every operation performed by the rewrite engine at the `notice` error level (see the `error_log` directive). Syntax: `on` or `off` Default value: `off` Example: `rewrite_log off;`

Table 4.6: A list of directives used to rewrite URIs

Common rewrite rules

Here is a set of rewrite rules that satisfy basic needs for dynamic websites that wish to beautify their page links, thanks to the URL rewriting mechanism. You will obviously need to adjust these rules according to your particular situation, as every website is different.

Performing a search

This rewrite rule is intended for search queries. Search keywords are included in the URL:

- **Input URI**: `http://website.com/search/some-search-keywords`
- **Rewritten URI**: `http://website.com/search.php?q=some-search-keywords`
- **Rewrite rule**: `rewrite ^/search/(.*)$ /search.php?q=$1?;`

User profile page

Most dynamic websites that allow visitors to register offer a profile view page. URLs of this form can be employed, containing both the user ID and the username:

- **Input URI**: `http://website.com/user/31/James`
- **Rewritten URI**: `http://website.com/user.php?id=31&name=James`
- **Rewrite rule**: `rewrite ^/user/([0-9]+)/(.+)$ /user.php?id=$1&name=$2?;`

Multiple parameters

Some websites may use different syntaxes for the argument string—for example, by separating non-named arguments with slashes:

- **Input URI**: `http://website.com/index.php/param1/param2/param3`
- **Rewritten URI**: `http://website.com/index.php?p1=param1&p2=param2&p3=param3`
- **Rewrite rule**: `rewrite ^/index.php/(.*)/(.*)/(.*)$ /index.php?p1=$1&p2=$2&p3=$3?;`

Wikipedia-like

Many websites have now adopted the URL style introduced by *Wikipedia*, including a prefix folder followed by an article name:

- **Input URI**: `http:// website.com/wiki/Some_keyword`
- **Rewritten URI**: `http://website.com/wiki/index.php?title=Some_keyword`
- **Rewrite rule**: `rewrite ^/wiki/(.*)$ /wiki/index.php?title=$1?;`

News website article

This URL structure is often employed by news websites, as URLs contain indications of articles' contents. It is formed of an article identifier, followed by a slash, then a list of keywords. The keywords can usually be ignored and not included in the rewritten URI:

- **Input URI**: `http://website.com/33526/us-economy-strengthens`
- **Rewritten URI**: `http://website.com/article.php?id=33526`
- **Rewrite rule**: `rewrite ^/([0-9]+)/.*$ /article.php?id=$1?;`

Discussion board

Modern bulletin boards now use *pretty URLs*, for the most part. This example shows how to create a *topic view* URL with two parameters: the topic identifier and the starting post. Once again, keywords are ignored:

- **Input URI**: `http://website.com/topic-1234-50-some-keywords.html`
- **Rewritten URI**: `http://website.com/viewtopic.php?topic=1234&start=50`
- **Rewrite rule**: `rewrite ^/topic-([0-9]+)-([0-9]+)-(.*)\.html$ /viewtopic.php?topic=$1&start=$2?;`

The first half of this chapter covered one of the most important NGINX modules—namely, the Rewrite module. There are a lot more modules that will greatly enrich the functionality of the web server; they are regrouped by theme in the following section.

Looking at some additional modules

Among the modules described in this section, some are included in the default NGINX build, but some are not. This implies that unless you specifically configured your NGINX build to include these modules (as described in *Chapter 1*), they will not be available to you. But remember that rebuilding NGINX to include additional modules is a relatively quick and easy process.

Website access and logging

The following set of modules allows you to configure how visitors access your website and the way your server logs requests.

Index

The Index module provides a simple directive named `index` that lets you define the page that NGINX will serve by default if no filename is specified in the client request (in other words, it defines the website `index` page). You may specify multiple filenames; the first file to be found will be served. If none of the specified files are found, NGINX will either attempt to generate an automatic index of the files, if the `autoindex` directive is enabled (check the HTTP `autoindex` module) or return a `403 Forbidden` error page.

Optionally, you may insert an absolute filename (such as `/page.html`), but only as the last argument of the directive.

Syntax: `index file1 [file2...] [absolute_file];`

Default value: `index.html`

Here's an example:

```
index index.php index.html index.htm;
index index.php index2.php /catchall.php;
```

This directive is valid in the following contexts: `http`, `server`, and `location`.

Autoindex

If NGINX cannot provide an index page for the requested directory, the default behavior is to return a `403 Forbidden` HTTP error page. With the following set of directives, you enable an automatic listing of the files that are present in the requested directory:

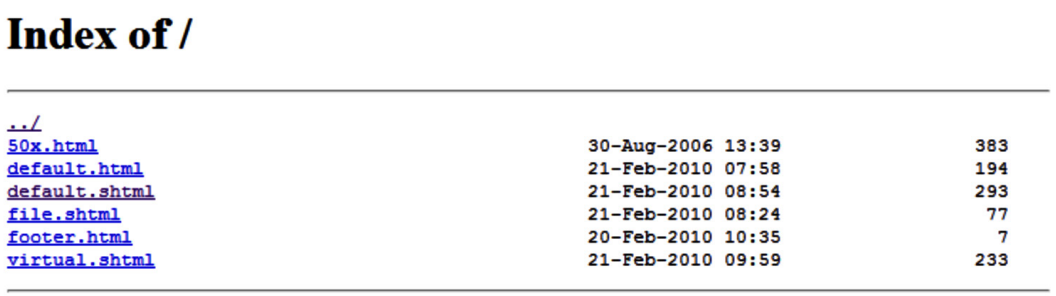


Figure 4.1: A screenshot showing the autoindex module listing the content of a directory

Three columns of information appear for each file: the filename, the file date and time, and the file size in bytes:

Directive	Description
`Autoindex` Context: `http`, `server`, `location`	Enables or disables automatic directory listing for directories missing an `index` page. Syntax: `on` or `off`
`autoindex_exact_size` Context: `http`, `server`, `location`	If set to `on`, this directive ensures that the listing displays file sizes in bytes. Otherwise, another unit is employed, such as KB, MB, or GB. Syntax: `on` or `off` Default value: `on`
`autoindex_localtime` Context: `http`, `server`, `location`	By default, this directive is set to `off`, so the date and time of files in the listing appear as GMT time. Set it to `on` to make use of the local server time. Syntax: `on` or `off` Default value: `off`
`autoindex_format` Context: `http`, `server`, `location`	NGINX offers to serve the directory index in different formats: HTML, XML, JSON, or JSONP (by default, HTML is used). Syntax: `autoindex_format html | xml | json | jsonp;` If you set the directive value to `jsonp`, NGINX will insert the value of the `callback` query argument as JSONP callback. For example, your script should call this URI: `/folder/?callback=MyCallbackName`.

Table 4.7: A list of directives for the autoindex module

Random index

This module enables a simple directive, `random_index`, which can be used within a `location` block in order for NGINX to return an `index` page selected randomly among the files of the specified directory.

> **Note**
> This module is not included in the default NGINX build.

Syntax: `on` or `off`

Log

This module controls the behavior of NGINX regarding access logs. It is a key module for system administrators, as it allows analyzing the runtime behavior of web applications. It is composed of three essential directives:

Directive	Description
`access_log` Context: `http`, `server`, `location`, `if` (in location), `limit_except`	This parameter defines the access log file path and the format of entries in the access log by selecting a template name or disables access logging. Syntax: `access_log path [format [buffer=size]] | off;` Some remarks concerning the directive syntax: • Use `access_log off` to disable access logging at the current level • The format argument corresponds to a template declared with the `log_format` directive • If the format argument is not specified, the default format is employed (combined) • You may use variables in the file path

`log_format`
 	
Context: `http`, `server`, `location`
	Defines a template to be utilized by the `access_log` directive, describing contents that should be included in an entry of the access log.
	
	Syntax: `log_format template_name [escape=default|json] format_string;`
	
	The default template is called `combined` and matches the following example:
	
	`log_format combined '$remote_addr - $remote_user [$time_local] '"$request" $status`
	
	`$body_bytes_sent '"$http_referer"`
	
	`"$http_user_agent"';`
	
	`# Other example`
	
	`log_format simple '$remote_addr $request';`

Directive	Description
`open_log_file_cache` Context: `http`, `server`, `location`	Configures the cache for log file descriptors. Please refer to the `open_file_cache` directive of the HTTP core module for additional information. Syntax: `open_log_file_cache max=N [inactive=time] [min_uses=N] [valid=time] | off;` The arguments are similar to the `open_file_cache` directive and other related directives, the difference being that this applies to access log files only.

Table 4.8: A list of directives for the Log module

The Log module also enables several new variables, though they are only accessible when writing log entries:

- `$connection`: The connection number
- `$pipe`: The variable is set to `p` if the request was pipelined
- `$time_local`: Local time (at the time of writing the log entry)
- `$msec`: Local time (at the time of writing the log entry) to the microsecond
- `$request_time`: Total length of the request processing, in milliseconds
- `$status`: Response status code
- `$bytes_sent`: Total number of bytes sent to the client
- `$body_bytes_sent`: Number of bytes sent to the client for the response body
- `$apache_bytes_sent`: Similar to `$body_bytes`, which corresponds to the `%B` parameter of Apache's `mod_log_config`
- `$request_length`: Length of the request body

Limits and restrictions

The following modules allow you to regulate access to the documents of your websites, require users to authenticate, match a set of rules, or simply restrict access to certain visitors.

auth_basic module

The `auth_basic` module enables the basic authentication functionality. With the two directives that it brings forth, you can make it so that a specific location of your website (or your server) is restricted to users who authenticate with a username and password:

```
location /admin/ {
    auth_basic "Admin control panel"; # variables are supported
    auth_basic_user_file access/password_file;
}
```

The first directive, `auth_basic`, can be set to either `off` or a text message, usually referred to as *authentication challenge* or *authentication realm*. This message is displayed by web browsers in a username/password box when a client attempts to access the protected resource.

The second one, `auth_basic_user_file`, defines the path of the password file relative to the directory of the configuration file. A password file is formed of lines respecting the `username:[{SCHEME}] password[:comment]` syntax:

- `username`: A plain text username.
- `{SCHEME}`: Optionally, the password hashing method. There are currently three supported schemes: `{PLAIN}` for plain-text passwords, `{SHA}` for SHA-1 hashing, and `{SSHA}` for salted SHA-1 hashing.

> **Important note**
>
> Please note that `PLAIN` and `SHA`, while still supported, are insecure methods. We recommend you use `SSHA` for salting passwords.

- `password`: The password.
- `comment`: A plain-text comment for your own use.

If you fail to specify a scheme, the password will need to be encrypted with the `crypt(3)` function—for example, with the help of the `htpasswd` command-line utility from Apache packages.

Access

Two important directives are brought up by this module: `allow` and `deny`. They let you allow or deny access to a resource for a specific IP address or IP address range.

Both directives have the same syntax: `allow IP | CIDR | unix: | all`, where `IP` is an IP address, `CIDR` is an IP address range (**Classless Inter-Domain Routing** or **CIDR** syntax), `unix:` represents all Unix domain sockets, and `all` specifies that the directive applies to all clients:

```
location {
allow 127.0.0.1; # allow local IP address
allow unix:; # allow UNIX domain sockets
    deny all; # deny all other IP addresses
}
```

Important note

Note that rules are processed from the top down: if your first instruction is `deny all`, all possible `allow` exceptions that you place afterward will have no effect. The opposite is also true; if you start with `allow all`, all possible `deny` directives that you place afterward will have no effect, as you already allowed all IP addresses.

Limit connections

The mechanism induced by this module is a little more complex than regular ones. It allows you to define the maximum amount of simultaneous connections to the server for a specific *zone*.

The first step is to define the zone using the `limit_conn_zone` directive:

- The directive syntax is `limit_conn_zone $variable zone=name:size;`
- `$variable` is the variable that will be used to differentiate one client from another, typically `$binary_remote_addr`, the IP address of the client in binary format (more efficient than ASCII)
- `name` is an arbitrary name given to the zone
- `size` is the maximum size you allocate to the table storing session states

The following example defines zones based on the client IP addresses (incoming connections):

```
limit_conn_zone $binary_remote_addr zone=myzone:10m;
```

Now that you have defined a zone, you may limit connections using `limit_conn`:

```
limit_conn zone_name connection_limit;
```

When applied to the previous example, it becomes the following:

```
location /downloads/ {
    limit_conn myzone 1;
}
```

As a result, requests that share the same `$binary_remote_addr` IP address are subject to the connection limit (one simultaneous connection). If the limit is reached, all additional concurrent requests will be answered with a `503 Service Unavailable` HTTP response. This response code can be overridden if you specify another code via the `limit_conn_status` directive. If you wish to log client requests that are affected by the limits you have set, enable the `limit_conn_log_level` directive and specify the log level (`info` | `notice` | `warn` | `error`).

Limit request

In a similar fashion, the Limit request module allows you to limit the amount of requests for a defined zone.

Defining the zone is done via the `limit_req_zone` directive; its syntax differs from the Limit zone equivalent directive:

```
limit_req_zone $variable zone=name:max_memory_size rate=rate;
```

The directive parameters are identical, except for the trailing `rate` parameter, expressed in **requests per second** (**r/s**) or **requests per minute** (**r/m**). It defines a request rate that will be applied to clients where the zone is enabled. To apply a zone to a location, use the `limit_req` directive:

```
limit_req zone=name burst=burst [nodelay];
```

The `burst` parameter defines the maximum possible bursts of requests. When the amount of requests received from a client exceeds the limit defined in the zone, responses are delayed in a manner that respects the rate that you defined. To a certain extent, only a maximum of `burst` requests will be accepted simultaneously. Past this limit, NGINX returns a `503 Service Unavailable` HTTP error response. This response code can be overridden if you specify another code via the `limit_req_status` directive:

```
limit_req_zone $binary_remote_addr zone=myzone:10m rate=2r/s;
[...]
location /downloads/ {
limit_req zone=myzone burst=10;
limit_req_status 404; # returns a 403 error if limit is exceeded
}
```

If you wish to log client requests that are affected by the limits you have set, enable the `limit_req_log_level` directive and specify the log level (`info` | `notice` | `warn` | `error`).

Auth request

The `auth_request` module was implemented in recent versions of NGINX and allows you to allow or deny access to a resource based on the result of a sub-request. NGINX calls the URI that you specify via the `auth_request` directive: if the sub-request returns a `2xx` response code (such as

`HTTP/200 OK`), access is allowed. If the sub-request returns a `401` or `403` status code, access is denied, and NGINX forwards the response code to the client. Should the backend return any other response code, NGINX will consider it to be an error and deny access to the resource:

```
location /downloads/ {
# if the script below returns a 200 status code,
# the download is authorized
auth_request /authorization.php;
}
```

Additionally, the module offers a second directive, called `auth_request_set`, allowing you to set a variable after the sub-request is executed. You can insert variables that originate from the sub-request upstream (`$upstream_http_*`), such as `$upstream_http_server` or other HTTP headers, from the server response:

```
location /downloads/ {
     # requests authorization from PHP script
     auth_request /authorization.php;
     # assuming authorization is granted, get filename from
     # sub-request response header and redirect
     auth_request_set $filename "${upstream_http_x_filename}.zip";
    rewrite ^ /documents/$filename;
}
```

Content and encoding

The following set of modules provides functionalities that have an effect on the contents served to the client, either by modifying the way the response is encoded, by affecting the headers, or by generating a response from scratch.

Empty GIF

The purpose of this module is to provide a directive that serves a 1x1 transparent GIF image from the memory. Such files are sometimes used by web designers to tweak the appearance of their websites. With this directive, you get an empty GIF straight from the memory, instead of reading and processing an actual GIF file from the storage space.

To utilize this feature, simply insert the `empty_gif` directive in the location of your choice:

```
location = /empty.gif {
    empty_gif;
}
```

MP4

MP4 is a module that enables a simple functionality that becomes useful when serving MP4 video files. It parses a special argument of the request, `start`, which indicates the offset of the section the client wishes to download or pseudo-stream. The video file must thus be accessed with the following URI: `video.mp4?start=XXX`. This parameter is prepared automatically by mainstream video players, such as JW Player.

> **Note**
> This module is not included in the default NGINX build.

To utilize this feature, simply insert the `.mp4` directive in the location of your choice:

```
location ~* \.mp4 {
    mp4;
}
```

Be aware that if NGINX fails to seek the requested position within the video file, the request will result in a `500 Internal Server Error` HTTP response. JW Player sometimes misinterprets this error and simply displays a `Video not found` error message.

HTTP headers

Three directives are introduced by this module that will affect the header of the response sent to the client.

First, `add_header name value [always]` lets you add a new line in the response headers, respecting the following syntax: `Name: value`. The line is added only for responses of the following codes: `200`, `201`, `204`, `301`, `302`, or `304`. You may insert variables in the `value` argument. If you specify `always` at the end of the directive value, the header will always be added, regardless of the response code.

Additionally, the `add_trailer name value [always]` directive allows you to add a header to the end of the response if the response code is one of `200`, `201`, `204`, `206`, `301`, `302`, `303`, `307`, or `308`. This directive can be specified multiple times to add multiple headers. The `always` flag works similarly to the `add_header` version.

Finally, the `expires` directive allows you to control the value of the `Expires` and `Cache-Control` HTTP headers sent to the client, affecting requests of the same code, as listed earlier. It accepts a single value among the following:

- `off`: Does not modify either header.
- **A time value**: The expiration date of the file is set to *the current time + the time you specify*. For example, `expires 24h` will return an expiry date set to 24 hours from now.

- `epoch`: The expiration date of the file is set to January 1, 1970. The `Cache-Control` header is set to `no-cache`.
- `max`: The expiration date of the file is set to December 31, 2037. The `Cache-Control` header is set to 10 years.

Addition

The Addition module allows you to (through simple directives) add content before or after the body of the HTTP response.

> **Note**
> This module is not included in the default NGINX build.

The two main directives are the following:

- `add_before_body file_uri;`
- `add_after_body file_uri;`

As stated previously, NGINX triggers a sub-request for fetching the specified URI. Additionally, you can define the types of files to which content is appended in case your `location` block pattern is not specific enough (default: `text/html`):

```
addition_types mime_type1 [mime_type2...];
addition_types *;
```

Substitution

Along the lines of the previous module, the Substitution module allows you to search and replace text directly from the response body:

```
sub_filter searched_text replacement_text;
```

This module is not included in the default NGINX build.

Two additional directives provide more flexibility:

- `sub_filter_once` (`on` or `off`, default `on`): Only replaces the text once, and stops after the first occurrence.
- `sub_filter_types` (default `text/html`): Affects additional MIME types that will be eligible for text replacement. The `*` wildcard is allowed.

Gzip filter

This module allows you to compress the response body with the Gzip algorithm before sending it to the client. To enable Gzip compression, use the `gzip` directive (`on` or `off`) at the `http`, `server`, `location`, and even the `if` level (though that is not recommended). The following directives will help you to further configure the filter options:

Directive	Description
`gzip_buffers` Context: `http`, `server`, `location`	Defines the amount and size of buffers to be used for storing the compressed response. Syntax: `gzip_buffers amount size;` Default: `gzip_buffers 4 4k` (or `8k`, depending on the OS)
`gzip_comp_level` Context: `http`, `server`, `location`	Defines the compression level of the algorithm. The specified value ranges from `1` (low compression, faster for the CPU) to `9` (high compression, slower). Syntax: Numeric value Default: `1`
`gzip_disable` Context: `http`, `server`, `location`	Disables Gzip compression for requests where the `User-Agent` HTTP header matches the specified regular expression. Syntax: Regular expression Default: None
`gzip_http_version` Context: `http`, `server`, `location`	Enables Gzip compression for the specified protocol version. Syntax: `1.0` or `1.1` Default: `1.1`
`gzip_min_length` Context: `http`, `server`, `location`	If the response body length is inferior to the specified value, it is not compressed. Syntax: Numeric value (size) Default: `0`

Directive	Description
`gzip_proxied` Context: `http`, `server`, `location`	Enables or disables Gzip compression for the body of responses received from a proxy (see reverse-proxying mechanisms in later chapters). The directive accepts the following parameters; some can be combined: • `off/any`: Disables or enables compression for all requests • `expired`: Enables compression if the `Expires` header prevents caching • `no-cache/no-store/private`: Enables compression if the `Cache-Control` header is set to `no-cache`, `no-store`, or `private` • `no_last_modified`: Enables compression in case the `Last-Modified` header is not set • `no_etag`: Enables compression in case the `ETag` header is not set • `auth`: Enables compression in case an `Authorization` header is set

`gzip_types`
 	
Context: `http`, `server`, `location`
	Enables compression for types other than the default `text/html` MIME type.
	
	Syntax:
	
	`gzip_types mime_type1 [mime_type2...];`
	
	`gzip_types *;`
	
	Default: `text/html` (cannot be disabled)

`gzip_vary` Context: `http`, `server`, `location`	Adds the `Vary: Accept-Encoding` HTTP header to the response. Syntax: `on` or `off` Default: `off`
`gzip_window` Context: `http`, `server`, `location`	Sets the size of the window buffer (`windowBits` argument) for Gzipping operations. This directive value is used for calls to functions from the Zlib library. Syntax: Numeric value (size) Default: `MAX_WBITS` constant from the Zlib library

Directive	Description
`gzip_hash` Context: `http`, `server`, `location`	Sets the amount of memory that should be allocated for the internal compression state (`memLevel` argument). This directive value is used for calls to functions from the Zlib library. Syntax: Numeric value (size) Default: `MAX_MEM_LEVEL` constant from the Zlib prerequisite library
`postpone_gzipping` Context: `http`, `server`, `location`	Defines a minimum data threshold to be reached before starting the Gzip compression. Syntax: Size (numeric value) Default: `0`
`gzip_no_buffer` Context: `http`, `server`, `location`	By default, NGINX waits until at least one buffer (defined by `gzip_buffers`) is filled with data before sending a response to the client. Enabling this directive disables buffering. Syntax: `on` or `off` Default: `off`

Table 4.9: A list of directives for the Gzip module

Gzip static

This module adds a simple functionality to the Gzip filter mechanism. When its `gzip_static` directive (`on`, `off`, or `always`) is enabled, NGINX will automatically look for a `.gz` file corresponding to the requested document before serving it. This allows NGINX to send pre-compressed documents instead of compressing documents on the fly at each request. Specifying `always` will force NGINX to serve the `gzip` version regardless of whether the client accepts `gzip` encoding.

> **Note**
> This module is not included in the default NGINX build.

If a client requests `/documents/page.html`, NGINX checks for the existence of a `/documents/page.html.gz` file. If the `.gz` file is found, it is served to the client. Note that NGINX does not generate `.gz` files itself, even after serving the requested files.

Gunzip filter

With the Gunzip filter module, you can decompress a Gzip-compressed response sent from a backend in order to serve it *raw* to the client—for example, in case the client browser is not able to process gzipped files (Microsoft Internet Explorer 6). Simply insert `gunzip on;` in a `location` block to employ this module. You can also set the buffer amount and size with `gunzip_buffers amount size;`, where `amount` is the amount of buffers to allocate and `size` is the size of each allocated buffer.

Charset filter

With the Charset filter module, you can control the character set of the response body more accurately. Not only are you able to specify the value of the `charset` argument of the `Content-Type` HTTP header (such as `Content-Type: text/html; charset=utf-8`), but NGINX can also re-encode data to a specified encoding method automatically:

Directive	Description
`charset` Context: `http`, `server`, `location`, `if`	This directive adds the specified encoding to the `Content-Type` header of the response. If the specified encoding differs from the `source_charset` one, NGINX re-encodes the document. Syntax: `charset encoding | off;` Default: `off` Example: `charset utf-8;`
`source_charset` Context: `http`, `server`, `location`, `if`	Defines the initial encoding of the response; if the value specified in the `charset` directive differs, NGINX re-encodes the document. Syntax: `source_charset encoding;`
`override_charset` Context: `http`, `server`, `location`, `if`	When NGINX receives a response from the proxy or FastCGI gateway, this directive defines whether or not the character encoding should be checked and potentially overridden. Syntax: `on` or `off` Default: `off`

Directive	Description

`charset_types`
Context: `http`, `server`, `location`
	Defines the MIME types that are eligible for re-encoding.
	Syntax:
	`charset_types mime_type1 [mime_type2...];`
	`charset_types * ;`
	Default: `text/html`, `text/xml`, `text/plain`, `text/vnd.wap.wml`, `application/x-javascript`, and `application/rss+xml`

`charset_map` Context: `http`	Lets you define character re-encoding tables. Each line of the table contains two hexadecimal codes to be exchanged. You will find re-encoding tables for the `koi8-r` character set in the default NGINX configuration folder (`koi-win` and `koi-utf`). Syntax: `charset_map src_encoding dest_encoding { ... }`

Table 4.10: A list of directives for the Charset filter module

Memcached

Memcached is a daemon application that can be connected via sockets. Its main purpose, as the name suggests, is to provide an efficient distributed key/value memory caching system. The NGINX Memcached module provides directives, allowing you to configure access to the Memcached daemon:

Directive	Description
`memcached_pass` Context: `location`, `if`	Defines the hostname and port of the Memcached daemon. Syntax: `memcached_pass hostname:port;` Example: `memcached_pass localhost:11211;`
`memcached_bind` Context: `http`, `server`, `location`	Forces NGINX to use the specified local IP address for connecting to the Memcached server. This can come in handy if your server has multiple network cards connected to different networks. Syntax: `memcached_bind IP_address;` Example: `memcached_bind 192.168.1.2;`

Directive	Description
`memcached_connect_timeout` Context: `http`, `server`, `location`	Defines the connection timeout in milliseconds (default: `60000`) Example: `memcached_connect_timeout 5000;`
`memcached_send_timeout` Context: `http`, `server`, `location`	Defines the data writing operations timeout in milliseconds (default: `60000`) Example: `memcached_send_timeout 5,000;`
`memcached_read_timeout` Context: `http`, `server`, `location`	Defines the data reading operations timeout in milliseconds (default: `60000`) Example: `memcached_read_timeout 5,000;`
`memcached_buffer_size` Context: `http`, `server`, `location`	Defines the size of the read and write buffer in bytes (default: page size) Example: `memcached_buffer_size 8k;`
`memcached_next_upstream` Context: `http`, `server`, `location`	When the `memcached_pass` directive is connected to an upstream block (see the Upstream module), this directive defines the conditions that should be matched in order to skip to the next upstream server. Syntax: Values selected among `error timeout`, `invalid_response`, `not_found`, or `off` Default: `error timeout` Example: `memcached_next_upstream off;`
`memcached_gzip_flag` Context: `http`, `server`, `location`	Checks for the presence of the specified flag in the `memcached` server response. If the flag is present, NGINX sets the `Content-encoding` header to `gzip` to indicate that it will be serving gzipped content. Syntax: Numeric flag Default: (none) Example: `memcached_gzip_flag 1;`

Table 4.11: A list of directives for the Memcached module

Additionally, you will need to define a `$memcached_key` variable that defines the key of the element that you are placing or fetching from the cache. You may, for instance, use `set $memcached_key $uri` or `set $memcached_key $uri?$args`.

Note that the NGINX Memcached module is only able to retrieve data from the cache; it does not store the results of requests. Storing data in the cache should be done by a server-side script. You just need to make sure to employ the same key naming scheme in both your server-side scripts and the NGINX configuration. As an example, we could decide to use `memcached` to retrieve data from the cache before passing the request to a proxy, if the requested URI is not found (see *Chapter 6*, for more details about the Proxy module):

```
server {
    server_name example.com;
    [...]
    location / {
        set $memcached_key $uri;
        memcached_pass 127.0.0.1:11211;
        error_page 404 @notcached;
    }
    location @notcached {
        internal;
        # if the file is not found, forward request to proxy
        proxy_pass 127.0.0.1:8080;
    }
}
```

Image filter

This module provides image processing functionalities through the **GD Graphics Library** (also known as **gdlib**).

> **Note**
> This module is not included in the default NGINX build.

Make sure to employ the following directives on a `location` block that filters image files only, such as `location ~* \.(png|jpg|gif|webp)$ { ... }`:

Directive	Description
`image_filter` Context: `location`	Lets you apply a transformation on the image before sending it to the client. There are five options available: • `off`: Turns off previously set `image_filter`. • `test`: Makes sure that the requested document is an image file; returns a `415 Unsupported Media Type` HTTP error if the test fails. • `size`: Composes a simple JSON response indicating information about the image, such as the size and type (for example; `{ "img": { "width":50, "height":50, "type":"png" } }`). If the file is invalid, a simple `{ }` instance is returned. • `resize width height`: Resizes the image to the specified dimensions. • `crop width height`: Selects a portion of the image of the specified dimensions. • `rotate 90 | 180 | 270`: Rotates the image by the specified angle (in degrees). Example: `image_filter resize 200 100;`
`image_filter_buffer` Context: `http`, `server`, `location`	Defines the maximum file size for images to be processed. Default: `image_filter_buffer 1m;`
`image_filter_jpeg_quality` Context: `http`, `server`, `location`	Defines the quality of output JPEG images. Default: `image_filter_jpeg_quality 75;`
`image_filter_webp_quality` Context: `http`, `server`, `location`	Defines the quality of output `webp` images. Default: `image_filter_webp_quality 80;`
`image_filter_transparency` Context: `http`, `server`, `location`	By default, PNG and GIF images keep their existing transparency during operations you perform using the Image filter module. If you set this directive to `off`, all existing transparency will be lost, but the image quality will be improved. Syntax: `on` or `off` Default: `on`

Directive	Description
`image_filter_sharpen` Context: `http`, `server`, `location`	Sharpens the image by a specified percentage (value may exceed 100). Syntax: Numeric value Default: `0`
`image_filter_interlace` Context: `http`, `server`, `location`	Enables interlacing of the output image. If the output image is a JPG file, the image is generated in *progressive JPEG* format. Syntax: `on` or `off` Default: `off`

Table 4.12: A list of directives for the Image filter module

> **Note**
> When it comes to JPG images, NGINX automatically strips off metadata (such as **EXIF**) if it occupies more than five percent of the total space of the file.

XSLT

The NGINX XSLT module allows you to apply an **Extensible Stylesheet Language Transformations** (**XSLT**) transform on an XML file or response received from a backend server (proxy, FastCGI, and so on) before serving the client:

Directive	Description
`xml_entities` Context: `http`, `server`, `location`	Specifies the document type definition (DTD) file containing symbolic element definitions. Syntax: File path Example: `xml_entities xml/entities.dtd;`
`xslt_stylesheet` Context: `location`	Specifies the XSLT template file path with its parameters. Variables may be inserted in the parameters. Syntax: `xslt_stylesheet template [param1] [param2...];` Example: `xslt_stylesheet xml/sch.xslt param=value;`

Directive	Description

`xslt_types`
 	
Context: `http`, `server`, `location`
	Defines additional MIME types to which transforms may apply, other than `text/xml`.
	
	Syntax: MIME type
	
	Example:
	
	`xslt_types  text/xml text/plain;`
	
	`xslt_types *;`

`xslt_paramxslt_string_param` Context: `http`, `server`, `location`	Both directives allow defining parameters for XSLT stylesheets. The difference lies in the way the specified value is interpreted: using `xslt_param`, XPath expressions in the value are processed; `xslt_string_param` should be used for plain character strings. Syntax: `xslt_param key value;`

Table 4.13: A list of directives for the XSLT module

About your visitors

The following set of modules provides extra functionality that will help you find out more information about visitors, such as by parsing client request headers for browser name and version, assigning an identifier to requests presenting similarities, and so on.

Browser

The Browser module parses the `User-Agent` HTTP header of the client request in order to establish values for variables that can be employed later in the configuration. The three variables produced are the following:

- `$modern_browser`: If the client browser is identified as being a modern web browser, the variable takes the value defined by the `modern_browser_value` directive
- `$ancient_browser`: If the client browser is identified as being an old web browser, the variable takes the value defined by `ancient_browser_value`
- `$msie`: This variable is set to `1` if the client is using a Microsoft Internet Explorer browser

To help NGINX recognize web browsers, distinguishing the old from the modern, you need to insert multiple occurrences of the `ancient_browser` and `modern_browser` directives:

```
modern_browser opera 10.0;
```

With this example, if the `User-Agent` HTTP header contains Opera 10.0, the client browser is considered modern.

Map

Just as with the Browser module, the Map module allows you to create maps of values, depending on a variable:

```
map $uri $variable {
  /page.html  0;
  /contact.html  1;
  /index.html  2;
  default 0;
}
rewrite ^ /index.php?page=$variable;
```

The `map` directive can only be inserted within the `http` block. Following this example, `$variable` may have three different values. If `$uri` was set to `/page.html`, the `$variable` value is now defined as `0`; if `$uri` was set to `/contact.html`, `$variable` is now `1`; if `$uri` was set to `/index.html`, the `$variable` value now equals `2`. For all other cases (`default`), `$variable` is set to `0`. The last instruction rewrites the URL accordingly. Apart from `default`, the `map` directive accepts another special keyword: `hostnames`. It allows you to match hostnames using wildcards such as `*.domain.com`. Finally, it's possible to mark a map as `volatile`, which makes the map non-cacheable.

Two additional directives allow you to tweak the way NGINX manages the mechanism in memory:

- `map_hash_max_size`: Sets the maximum size of the hash table holding a map
- `map_hash_bucket_size`: The maximum size of an entry in the map

Regular expressions may also be used in patterns if you prefix them with `~` (case-sensitive) or `~*` (case-insensitive):

```
map $http_referer $ref {
~google "Google";
    ~* yahoo "Yahoo";
    \~bing "Bing"; # not a regular expression due to the \ before the
tilde
default $http_referer; # variables may be used
}
```

Geo

The purpose of this module is to provide functionality that is quite similar to the `map` directive, affecting a variable based on client data (in this case, the IP address). The syntax is slightly different, in the extent that you are allowed to specify IPv4 and IPv6 address ranges (in CIDR format):

```
geo $variable {
  default unknown;
  127.0.0.1   local;
  123.12.3.0/24  uk;
  92.43.0.0/16  fr;
}
```

Note that the preceding block is being presented to you just for the sake of the example and does not actually detect UK and French visitors; you'll want to use the GeoIP module if you wish to achieve proper geographical location detection. In this block, you may insert a number of directives that are specific to this module:

- `delete`: Allows you to remove the specified subnetwork from the mapping.
- `default`: The default value given to `$variable` in case the user's IP address does not match any of the specified IP ranges.
- `include`: Allows you to include an external file.
- `proxy`: Defines a subnet of trusted addresses. If the user IP address is among those trusted, the value of the `X-Forwarded-For` header is used as the IP address instead of the socket IP address.
- `proxy_recursive`: If enabled, this will look for the value of the `X-Forwarded-For` header, even if the client IP address is not trusted.
- `ranges`: If you insert this directive as the first line of your `geo` block, it allows you to specify IP ranges instead of CIDR masks. The following syntax is thus permitted: `127.0.0.1-127.0.0.255 LOCAL;`.

GeoIP

Although the name suggests some similarities with the previous one, this optional module provides accurate geographical information about your visitors by making use of *MaxMind* (`https://www.maxmind.com/en/home`) GeoIP binary databases. You need to download the database files from the *MaxMind* website and place them in your NGINX directory.

> **Note**
> This module is not included in the default NGINX build.

Then, all you have to do is specify the database path with either directive:

```
geoip_country country.dat; # country information db
geoip_city city.dat; # city information db
geoip_org geoiporg.dat; # ISP/organization db
```

The first directive enables several variables: `$geoip_country_code` (two-letter country code), `$geoip_country_code3` (three-letter country code), and `$geoip_country_name` (full country name). The second directive includes the same variables but provides additional information: `$geoip_region`, `$geoip_city`, `$geoip_postal_code`, `$geoip_city_continent_code`, `$geoip_latitude`, `$geoip_longitude`, `$geoip_dma_code`, `$geoip_area_code`, and `$geoip_region_name`. The third directive offers information about the organization or ISP that owns the specified IP address, by filling up the `$geoip_org` variable.

> **Note**
>
> If you need the variables to be encoded in UTF-8, simply add the `utf8` keyword at the end of the `geoip_` directives.

UserID filter

This module assigns an identifier to clients by issuing cookies. The identifier can be accessed from the `$uid_got` and `$uid_set` variables further in the configuration:

Directive	Description
`userid` Context: `http`, `server`, `location`	Enables or disables issuing and logging of cookies. The directive accepts four possible values: • `on`: Enables v2 cookies and logs them • `v1`: Enables v1 cookies and logs them • `log`: Does not send cookie data but logs incoming cookies • `off`: Does not send cookie data Default value: `userid off;`
`userid_service` Context: `http`, `server`, `location`	Defines the IP address of the server issuing the cookie. Syntax: `userid_service ip;` Default: IP address of the server

Directive	Description
`userid_name` Context: `http`, `server`, `location`	Defines the name assigned to the cookie. Syntax: `userid_name name;` Default value: The user identifier
`userid_domain` Context: `http`, `server`, `location`	Defines the domain assigned to the cookie. Syntax: `userid_domain domain;` Default value: None (the domain part is not sent)
`userid_path` Context: `http`, `server`, `location`	Defines the path part of the cookie. Syntax: `userid_path path;` Default value: `/`
`userid_expires` Context: `http`, `server`, `location`	Defines the cookie expiration date. Syntax: `userid_expires date | max;` Default value: No expiration date
`userid_p3p` Context: `http`, `server`, `location`	Assigns a value to the **Platform for Privacy Preferences Project** (**P3P**) header sent with the cookie. Syntax: `userid_p3p data;` Default value: None

Table 4.14: A list of directives for the userid cookie identifier module

Referer

A simple directive is introduced by this module: `valid_referers`. Its purpose is to check the `Referer` HTTP header from the client request and to possibly deny access based on the value. If the referrer is considered invalid, `$invalid_referer` is set to `1`. In the list of valid referrers, you may employ three kinds of values:

- **None**: The absence of a referrer is considered to be a valid referrer
- **Blocked**: A masked referrer (such as `XXXXX`) is also considered valid
- **A server name**: The specified server name is considered to be a valid referrer

Following the definition of the `$invalid_referer` variable, you may, for example, return an error code if the referrer was found invalid:

```
valid_referers none blocked *.website.com *.google.com;
  if ($invalid_referer) {
  return 403;
}
```

Be aware that spoofing the `Referer` HTTP header is a very simple process, so checking the referer of client requests should not be used as a security measure.

Two more directives are offered by this module, `referer_hash_bucket_size` and `referer_hash_max_size`, respectively allowing you to define the bucket size and maximum size of valid referrers' hash tables.

Real IP

This module provides one simple feature: it replaces the client IP address with the one specified in the `X-Real-IP` HTTP header for clients that visit your website behind a proxy or for retrieving IP addresses from the proper header if NGINX is used as a backend server (it essentially has the same effect as Apache's `mod_rpaf` if you have experience with Apache). To enable this feature, you need to insert the `real_ip_header` directive that defines the HTTP header to be exploited: either `X-Real-IP` or `X-Forwarded-For`. The second step is to define trusted IP addresses; in other words, the clients that are allowed to make use of those headers. This can be done thanks to the `set_real_ip_from` directive, which accepts both IP addresses and CIDR address ranges:

```
real_ip_header X-Forwarded-For;
set_real_ip_from 192.168.0.0/16;
set_real_ip_from 127.0.0.1;
set_real_ip_from unix:; # trusts all UNIX-domain sockets
```

Directive	Description
`set_real_ip_from` Context: `http`, `server`, `location`	Sets the trusted addresses that will trigger the real IP header replacement. Set this to the IP of the trusted reverse proxy (or proxies) in front of NGINX. This directive can be specified multiple times, and hostnames are allowed. The special value of `unix:` sets all Unix sockets as trusted. Syntax: `set_real_ip_from address | CIDR | unix:;` Default: None

Directive	Description
`real_ip_header` Context: `http`, `server`, `location`	Sets the header field that will be used as a replacement for the IP address. The `proxy_protocol` special value changes the IP to the one from the Proxy protocol. Syntax: `real_ip_header field | X-Real-Ip | X-Forwarded-For | proxy_protocol;` Default: `X-Real-Ip`
`real_ip_recursive` Context: `http`, `server`, `location`	If set to `on`, the replacement IP will be set to the last non-trusted IP in the `real_ip_header` field. If set to `off`, will be replaced with the last IP in the `real_ip_header` field, whether trusted or not. Syntax: `on` or `off` Default: `off`

Table 4.15: A list of directives for the real_ip module

> **Note**
> This module is not included in the default NGINX build.

SSL and security

NGINX provides secure HTTP functionalities through the SSL module, but also offers an extra module, called **Secure Link**, that helps you protect your website and visitors in a totally different way.

SSL

The SSL module enables HTTPS support, HTTP over SSL/TLS in particular. It gives you the possibility to serve secure websites by providing a certificate, a certificate key, and other parameters, defined with the following directives:

Directive	Description
`ssl` Context: `http`, `server`	Enables HTTPS for the specified server. This directive is the equivalent of `listen 443 ssl` or `listen port ssl`, more generally. Syntax: `on` or `off` Default: `ssl off;`

Directive	Description
`ssl_certificate` Context: `http`, `server`	Sets the path of the Privacy Enhanced Mail (PEM) certificate. This directive can be specified multiple times to load certificates of different types. Syntax: File path
`ssl_certificate_key` Context: `http`, `server`	Sets the path of the PEM secret key file. This directive can be specified multiple times to load certificates of different types. Syntax: File path
`ssl_client_certificate` Context: `http`, `server`	Sets the path of the client PEM certificate. Syntax: File path
`ssl_crl` Context: `http`, `server`	Orders NGINX to load a Certificate Revocation List (CRL) file, which allows checking the revocation status of certificates when using TLS mutual authentication.
`ssl_dhparam` Context: `http`, `server`	Sets the path of the Diffie-Hellman parameters file. Syntax: File path
`ssl_protocols` Context: `http`, `server`	Specifies the protocol that should be employed. Syntax: `ssl_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2] [TLSv1.3];` Default: `ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;`
`ssl_ciphers` Context: `http`, `server`	Specifies the ciphers that should be employed. A list of available ciphers can be obtained by running the following command from the shell: `openssl ciphers`. Syntax: `ssl_ciphers cipher1[:cipher2...];` Recommended: Please visit `ssl-config.mozilla.org` for the default ciphers to use.
`ssl_prefer_server_ciphers` Context: `http`, `server`	Specifies whether server ciphers should be preferred over client ciphers. Syntax: `on` or `off` Default: `off`

Directive	Description
`ssl_verify_client` Context: `http`, `server`	Enables verifying certificates transmitted by the client and sets the result in `$ssl_client_verify`. The `optional_no_ca` value verifies the certificate if there is one, but does not require it to be signed by a trusted certificate authority (CA) certificate. Syntax: `on | off | optional | optional_no_ca` Default: `off`
`ssl_verify_depth` Context: `http`, `server`	Specifies the verification depth of the client certificate chain. Syntax: Numeric value Default: `1`
`ssl_session_cache` Context: `http`, `server`	Configures the cache for SSL sessions. Syntax: `off`, `none`, `builtin:size`, or `shared:name:size` Default: `off` (disables SSL sessions)
`ssl_session_timeout` Context: `http`, `server`	When SSL sessions are enabled, this directive defines the timeout for using session data. Syntax: Time value Default: 5 minutes
`ssl_password_phrase` Context: `http`, `server`	Specifies a file containing passphrases for secret keys. Each passphrase is specified on a separate line; they are tried one after the other when loading a certificate key. Syntax: Filename Default: (none)
`ssl_buffer_size` Context: `http`, `server`	Specifies buffer size when serving requests over SSL. Syntax: Size value Default: `16k`

Directive	Description
`ssl_session_tickets` Context: `http`, `server`	Enables TLS session tickets, allowing for the client to reconnect faster, skipping renegotiation. Syntax: `on` or `off` Default: `on`
`ssl_session_ticket_key` Context: `http`, `server`	Sets the path of the key file used to encrypt and decrypt TLS session tickets. By default, a random value is generated. Syntax: Filename Default: (none)
`ssl_trusted_certificate` Context: `http`, `server`	Sets the path of a trusted certificate file (PEM format), used to validate the authenticity of client certificates, as well as stapling of Online Certificate Status Protocol (OCSP) responses. More about SSL stapling can be found in the next section. Syntax: Filename Default: (none)

Table 4.16: A list of directives for the SSL module

Note
This module is not included in the default NGINX build.

Additionally, the following variables are made available:

- `$ssl_cipher`: Indicates the cipher used for the current request
- `$ssl_ciphers`: Returns a list of client-supported ciphers
- `$ssl_curves`: Returns a list of client-supported curves for Elliptic Curves/EC/ECDH
- `$ssl_client_serial`: Indicates the serial number of the client certificate
- `$ssl_client_s_dn` and `$ssl_client_i_dn`: Indicates the value of the subject and issuer **Distinguished Name** (**DN**) of the client certificate
- `$ssl_protocol`: Indicates the protocol at use for the current request
- `$ssl_client_cert` and `$ssl_client_raw_cert`: Returns client certificate data, which is raw data for the second variable

- `$ssl_client_verify`: Set to `SUCCESS` if the client certificate was successfully verified
- `$ssl_session_id`: Allows you to retrieve the ID of an SSL session
- `$ssl_client_escaped_cert`: Returns the client certificate in PEM format

Setting up an SSL certificate

Although the SSL module offers a lot of possibilities, in most cases, only a couple of directives are actually useful for setting up a secure website. This guide will help you configure NGINX to use an SSL certificate for your website (in the example, your website is identified by `secure.website.com`). Before doing so, ensure that you already have the following elements at your disposal:

- A `.key` file generated with the following command: `openssl genrsa -out secure.website.com.key 2048` (other encryption levels work too)
- A `.csr` file generated with the following command: `openssl req -new -key secure.website.com.key -out secure.website.com.csr`
- Your website certificate file, as issued by the CA; for example, `secure.website.com.crt` (in order to obtain a certificate from the CA, you will need to provide your `.csr` file)
- The CA certificate file as issued by the CA (for example, `gd_bundle.crt`, if you purchased your certificate from `https://godaddy.com/`)

The first step is to merge your website certificate and the CA certificate together with the following command:

```
cat secure.website.com.crt gd_bundle.crt > combined.crt
```

You are then ready to configure NGINX to serve secure content:

```
server {
    listen 443;
    server_name secure.website.com;
    ssl on;
    ssl_certificate /path/to/combined.crt;
    ssl_certificate_key /path/to/secure.website.com.key;
    [...]
}
```

SSL stapling

SSL stapling, also called OCSP stapling, is a technique that allows clients to easily connect and resume sessions to an SSL/TLS server without having to contact the CA, thus reducing SSL negotiation times. In normal OCSP transactions, the client contacts the CA to check the revocation status of the server's certificate. In the case of high-traffic websites, this can cause huge stress on CA servers. An

intermediary solution was designed: stapling. The OCSP record is obtained from the CA by your server itself periodically and **stapled** to exchanges with the client. The OCSP record is cached by your server for a period of up to 48 hours in order to limit communications with the CA.

Enabling SSL stapling should thus speed up communications between your visitors and your server. Achieving this in NGINX is relatively simple—all you really need is to insert three directives in your `server` block and obtain a fully trusted certificate chain file (containing both the root and intermediate certificates) from your CA:

- `ssl_stapling on`: Enables SSL stapling within the `server` block
- `ssl_stapling_verify on`: Enables verification of OCSP responses by the server
- `ssl_trusted_certificate filename`: Here, `filename` is the path of your full trusted certificate file (extension should be `.pem`)

Two optional directives also exist, allowing you to modify the behavior of this module:

- `ssl_stapling_file filename`: Here, `filename` is the path of a cached OCSP record, overriding the record provided by the OCSP responder specified in the certificate file
- `ssl_stapling_responder url`: Here, `url` is the URL of your CA's OCSP responder, overriding the URL specified in the certificate file

If you are having issues connecting to the OCSP responder, make sure your NGINX configuration contains a valid DNS resolver using the `resolver` directive (a local DNS responder might help too).

Other miscellaneous and optional modules

The remaining two modules (which all need to be enabled at compile time) are optional and provide additional advanced functionalities.

Stub status

The **Stub status** module was designed to provide information about the current state of the server, such as the number of active connections, total handled requests, and more. To activate it, place the `stub_status` directive in a `location` block. All requests matching the `location` block will produce a status page:

```
location = /nginx_status {
    stub_status on;
    allow 127.0.0.1; # you may want to protect the information
    deny all;
}
```

> **Note**
> This module is not included in the default NGINX build. Make sure to protect this page as it should not be used in production.

Here's an example result produced by NGINX:

```
Active connections: 1
server accepts handled requests
10 10 23
Reading: 0 Writing: 1 Waiting: 0
```

It's interesting to note that there are several server monitoring solutions, such as *netdata*, that offer NGINX support through the Stub status page by calling it at regular intervals and parsing the statistics.

Degradation

The HTTP Degradation module configures your server to return an error page when your server runs low on memory. It works by defining a memory amount that is to be considered low, and then specifying locations for which you wish to enable a degradation check:

```
degradation sbrk=500m; # to be inserted at the http block level
degrade 204; # in a location block, specify the error code (204 or
444) to return in case the server condition has degraded
```

Integrating a third-party module into your NGINX build

The NGINX community has been growing larger over the past few years, and many additional modules have been written by third-party developers. These can be downloaded from the official wiki website at `https://www.nginx.com/resources/wiki/modules/`. However, we advise you to be careful when integrating these third-party modules as they do not come with security support or assurances from the NGINX development team, potentially introducing security vulnerabilities.

The currently available modules offer a wide range of new possibilities, among which are the following:

- An *Access Key* module to protect your documents in a similar fashion to Secure Link, by Mykola Grechukh
- A *Fancy Indexes* module that improves the automatic directory listings generated by NGINX, by Adrian Perez de Castro
- The *Headers More* module that improves flexibility with HTTP headers, by Yichun Zhang (`agentzh`)
- Many more features for various parts of the web server

To integrate a third-party module into your NGINX build, you need to follow these three simple steps:

1. Download the `.tar.gz` archive associated with the module you wish to download.
2. Extract the archive with the following command:

```
tar xzf module.tar.gz
```

3. Configure your NGINX build with the following command:

```
./configure --add-module=/module/source/path [...]
```

Once you have finished building and installing the application, the module is available just like a regular NGINX module, with its directives and variables.

If you are interested in writing NGINX modules yourself, Evan Miller published an excellent walkthrough: *Emiller's Guide to Nginx Module Development*. The complete guide may be consulted from his personal website at `https://www.evanmiller.org/`.

Summary

Throughout this chapter, we have been discovering modules that help you improve or fine-tune the configuration of your web server. NGINX fiercely stands up to other concurrent web servers in terms of functionality, and its approach to virtual hosts and the way they are configured will probably convince many administrators to make the switch.

Three additional modules were left out, though. Firstly, the FastCGI module will be approached in the next chapter, as it will allow us to configure a gateway to applications such as PHP or Python. Secondly, the Proxy module that lets us design complex setups will be described in *Chapter 6*. Finally, the Upstream module is tied to both, so it will be detailed in parallel.

5
PHP and Python with NGINX

The 2000s have been the decade of server-side technologies. Over the past 15 years or so, an overwhelming majority of websites have migrated from simple static HTML content to highly and fully dynamic pages, taking the web to an entirely new level in terms of interaction with visitors. Software solutions emerged quickly, including open source ones, and some became mature enough to process high-traffic websites. In this chapter, we will study the ability of NGINX to interact with these applications. We have selected two for different reasons. The first one is obviously PHP. As of June 2015, *W3Techs* (a website specializing in web technology surveys) reveals that PHP empowers over 80% of websites designed with a server-side language. The second language in our selection is Python, due to the way it is installed and configured to work with NGINX. The mechanism we will discover effortlessly applies to other applications, such as Perl or **Ruby on Rails** (**RoR**).

This chapter covers the following topics:

- Introduction to **Fast Common Gateway Interface** (**FastCGI**) technologies
- Setting up NGINX with PHP and **PHP FastCGI Process Manager** (**PHP-FPM**)
- Setting up NGINX with Python and Django

Introduction to FastCGI technologies

Before we begin, you should know that (as the name suggests) **FastCGI** is actually a variation of **Common Gateway Interface** (**CGI**). Therefore, explaining CGI first is in order. The improvements introduced by FastCGI are detailed in the following sections.

Understanding the CGI mechanism

The original purpose of a web server was merely to respond to requests from clients by serving files located on a storage device. The client sends a request to download a file, and the server processes the request and sends the appropriate response: **200 OK** if the file can be served normally, 404 if the file was not found, and other variants, as illustrated in the following diagram:

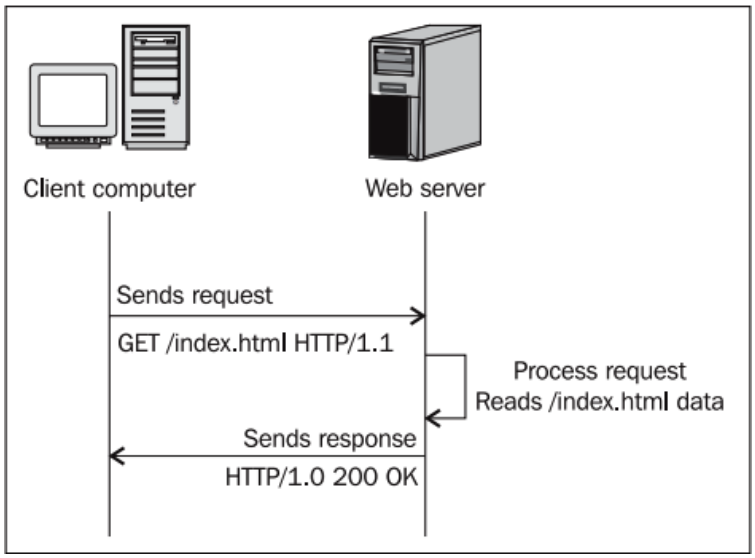


Figure 5.1: A diagram depicting standard HTTP requests

This mechanism has been in use since the beginning of the World Wide Web, and it still is. However, as stated before, static websites are being progressively abandoned at the expense of dynamic ones that contain scripts processed by applications such as PHP and Python, among others. The web-serving mechanism thus evolved into the following:

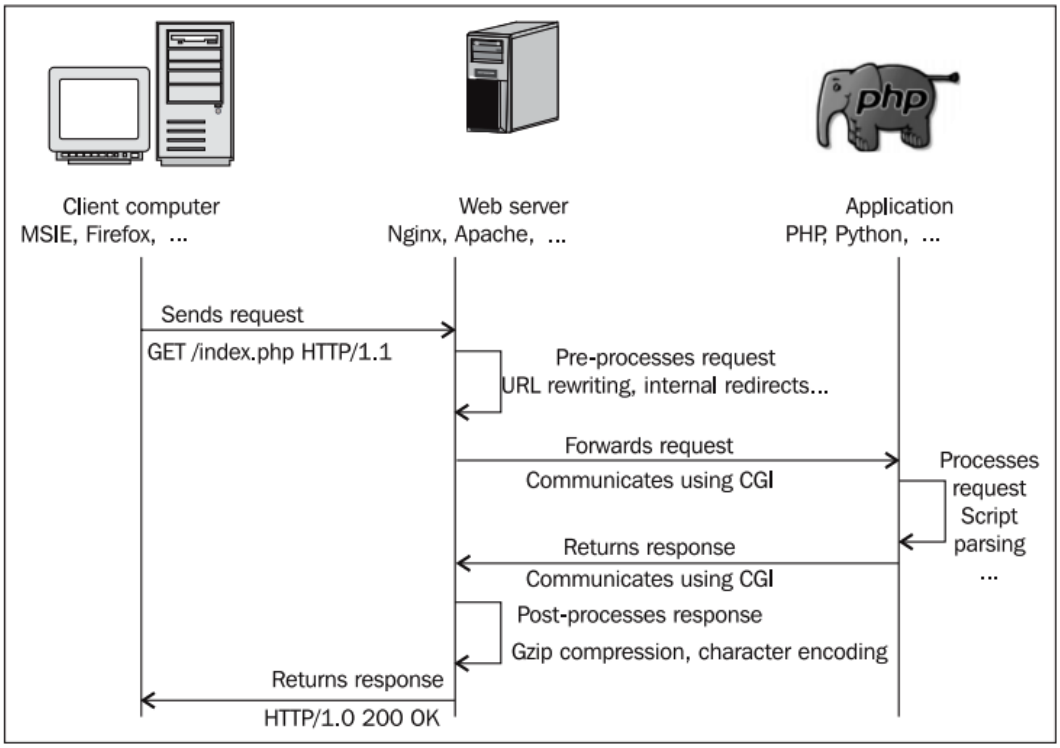


Figure 5.2: A diagram depicting standard HTTP requests with Nginx and CGI processing

When a client attempts to visit a dynamic page, the web server receives the request and forwards it to a third-party application. The application processes the script independently and returns the produced response to the web server, which then forwards the response back to the client.

In order for the web server to communicate with that application, the CGI protocol was invented in the early 1990s.

CGI

The following is stated in *RFC 3875 (CGI protocol v1.1)*, designed by the **Internet Society** (**ISOC**):

The CGI allows an HTTP server and a CGI script to share responsibility for responding to client requests. [...]. The server is responsible for managing connection, data transfer, transport, and network issues related to the client request, whereas the CGI script handles the application issues such as data access and document processing.

CGI is the protocol that describes the way information is exchanged between the web server (NGINX) and the gateway application (PHP, Python, and so on). In practice, when the web server receives a request that should be forwarded to the gateway application, it simply executes the command corresponding to the desired application; for example, `/usr/bin/php`. Details about the client request (such as the `User-Agent` header and other request information) are passed either as command-line arguments or in environment variables, while actual data from `POST` or `PUT` requests is transmitted through the standard input. The invoked application then writes the processed document contents to the standard output, which is recaptured by the web server.

While this technology seems simple and efficient enough at first sight, it comes with a few major drawbacks, which are discussed as follows:

- A unique process is spawned for each request. Memory and other context information are lost from one request to another.
- Starting up a process can be resource-consuming for the system. Massive numbers of simultaneous requests (each spawning a process) could quickly clutter a server.
- Designing an architecture where the web server and the gateway application are located on different computers seems difficult, if not impossible.

FastCGI

The issues mentioned in the *CGI* section render the CGI protocol relatively inefficient for servers that are subject to heavy load. The will to find solutions led the open market in the mid-90s to develop an evolution of CGI: FastCGI. It has become a major standard over the past 15 years, and most web servers now offer the functionality, even proprietary server software such as Microsoft **Internet Information Services** (**IIS**).

Although the purpose remains the same, FastCGI offers significant improvements over CGI with the establishment of the following principles:

- Instead of spawning a new process for each request, FastCGI employs persistent processes that come with the ability to handle multiple requests.
- The web server and the gateway application communicate with the use of sockets such as TCP or Unix local IPC sockets. Consequently, the web server and backend processes may be located on two different computers on a network.
- The web server forwards the client request to the gateway and receives a response within a single connection. Additional requests may also follow without needing to create additional connections. Note that on most web servers, including NGINX and Apache, the implementation of FastCGI does not (or at least not fully) support *multiplexing*.
- Since FastCGI is a socket-based protocol, it can be implemented on any platform with any programming language.

Throughout this chapter, we will be setting up PHP and Python via FastCGI. Additionally, you will find the mechanism to be relatively similar in the case of other applications, such as Perl or RoR.

Designing a FastCGI-powered architecture is actually not as complex as one might imagine. As long as you have the web server and the backend application running, the only difficulty that remains is to establish a connection between both parties. The first step in that perspective is to configure the way NGINX will communicate with the FastCGI application. FastCGI compatibility with NGINX is introduced by the FastCGI module, which is included in default NGINX builds (including those that are installed via software repositories). This section details the directives that are made available by the module.

uWSGI and SCGI

Before reading the rest of the chapter, you should know that NGINX offers two other CGI-derived module implementations:

- The `uWSGI` module allows NGINX to communicate with applications through the `uwsgi` protocol, itself derived from the **Web Server Gateway Interface** (**WSGI**). The most commonly used (the unique) server implementing the `uwsgi` protocol is the unoriginally named uWSGI server. Its latest documentation can be found at `http://uwsgi-docs.readthedocs.io/en/latest/`. This module will prove useful to Python adepts, seeing as the uWSGI project was designed mainly for Python applications.
- **SCGI**, which stands for **Simple Common Gateway Interface**, is a variant of the CGI protocol, much like FastCGI. Younger than FastCGI since its specification was first published in 2006, SCGI was designed to be easier to implement and as its name suggests: simple. It is not related to a particular programming language. SCGI interfaces and modules can be found in a variety of software projects such as Apache, IIS, Java, Cherokee, and a lot more.

There are no major differences in the way NGINX handles the FastCGI, uWSGI, and SCGI protocols: each of these has its respective module, containing similarly named directives. The following table lists a couple of directives from the FastCGI module, which are detailed in the following sections, and their uWSGI and SCGI equivalents:

FastCGI module	uWSGI equivalent	SCGI equivalent
`fastcgi_pass`	`uwsgi_pass`	`scgi_pass`
`fastcgi_cache`	`uwsgi_cache`	`scgi_cache`
`fastcgi_temp_path`	`uwsgi_temp_path`	`scgi_temp_path`

Table 5.1: FastCGI, uWSGI, and SCGI equivalent directives

Directive names and syntaxes are identical. In addition, the NGINX development team has been maintaining all three modules in parallel. New directives or directive updates are always applied to

all of them. As such, the following sections will document NGINX's implementation of the FastCGI protocol (since it is the most widely used), but they also apply to uWSGI and SCGI.

Main directives

The FastCGI, uWSGI, and SCGI modules are included in the default NGINX build. You do not need to enable them manually at compile time. The directives listed in the following table allow you to configure the way NGINX *passes* requests to the FastCGI/uWSGI/SCGI application. Note that you will find `fastcgi_params`, `uwsgi_params`, and `scgi_params` files in the NGINX configuration folder; these define directive values that are valid for most situations:

Directive	Description

`fastcgi_pass`
 	
Context: `location`, `if`
	This directive specifies that the request should be passed to the FastCGI server, by indicating its location:
	• For TCP sockets, the syntax is `fastcgi_pass hostname:port;`
	• For Unix domain sockets, the syntax is `fastcgi_pass unix:/path/to/fastcgi.socket;`
	• You may also refer to upstream blocks (read the following sections for more information): `fastcgi_pass myblock;`
	
	Examples:
	`fastcgi_pass localhost:9000;`
	`fastcgi_pass 127.0.0.1:9000;`
	`fastcgi_pass unix:/tmp/fastcgi.socket;`
	`# Using an upstream block`
	`upstream fastcgi {`
	`server 127.0.0.1:9000;`
	`server 127.0.0.1:9001;`
	`}`
	`location ~* \.php$ {`
	`fastcgi_pass fastcgi;`
	`}`

Directive	Description

`fastcgi_param`
 	
Context: `http`, `server`, `location`
	This directive allows you to configure the request passed to FastCGI. Two parameters are strictly required for all FastCGI requests: `SCRIPT_FILENAME` and `QUERY_STRING`.
	
	Examples:
	
	`fastcgi_param SCRIPT_FILENAME /home/website.com/www$fastcgi_script_name;`
	
	`fastcgi_param QUERY_STRING $query_string;`
	
	As for `POST` requests, additional parameters are required: `REQUEST_METHOD`, `CONTENT_TYPE`, and `CONTENT_LENGTH`.
	
	Examples:
	
	`fastcgi_param REQUEST_METHOD $request_method;`
	
	`fastcgi_param CONTENT_TYPE $content_type;`
	
	`fastcgi_param CONTENT_LENGTH $content_length;`
	
	The `fastcgi_params` file that you will find in the NGINX configuration folder already includes all of the necessary parameter definitions, except for `SCRIPT_FILENAME`, which you need to specify for each of your FastCGI configurations.
	
	If the parameter name begins with `HTTP_`, it will override potentially existing HTTP headers of the client request.
	
	You may optionally specify the `if_not_empty` keyword, forcing NGINX to transmit the parameter only if the specified value is not empty.
	
	Syntax: `fastcgi_param PARAM value [if_not_empty];`

`fastcgi_bind` Context: `http`, `server`, `location`	This directive binds the socket to a local IP address, allowing you to specify the network interface you want to use for FastCGI communications. Syntax: `fastcgi_bind IP_address[:port] [transparent] | off;`
`fastcgi_pass_header` Context: `http`, `server`, `location`	This directive specifies additional headers that should be passed to the FastCGI server. Syntax: `fastcgi_pass_header headername;` Example: `fastcgi_pass_header Authorization;`

Directive	Description
`fastcgi_hide_header` Context: `http`, `server`, `location`	This directive specifies headers that should be hidden from the FastCGI server (headers that NGINX does not forward). Syntax: `fastcgi_hide_header headername;` Example: `fastcgi_hide_header X-Forwarded-For;`
`fastcgi_index` Context: `http`, `server`, `location`	The FastCGI server does not support automatic directory indexes. If the requested URI ends with `/`, NGINX appends the `fastcgi_index` value. Syntax: `fastcgi_index filename;` Example: `fastcgi_index index.php;`
`fastcgi_ignore_client_abort` Context: `http`, `server`, `location`	This directive lets you define what happens if the client aborts their request to the web server. If the directive is turned on, NGINX ignores the abort request and finishes processing the request. If it's turned off, NGINX does not ignore the abort request. It interrupts the request treatment and aborts related communication with the FastCGI server. Syntax: `on` or `off` Default: `off`
`fastcgi_intercept_errors` Context: `http`, `server`, `location`	This directive defines whether or not NGINX should process errors returned by the gateway or directly return error pages to the client. Error processing is done via the `error_page` directive of NGINX. Syntax: `on` or `off` Default: `off`
`fastcgi_read_timeout` Context: `http`, `server`, `location`	This directive defines the timeout for the response from the FastCGI application. If NGINX does not receive the response after this period, a `504 Gateway Timeout` HTTP error is returned. Syntax: Numeric value (in seconds) Default: `60`

Directive	Description
`fastcgi_connect_timeout` Context: `http`, `server`, `location`	This directive defines the backend server connection timeout. This is different than the read/send timeout. If NGINX is already connected to the backend server, the `fastcgi_connect_timeout` directive is not applicable. Syntax: Time value (in seconds) Default: `60`
`fastcgi_send_timeout` Context: `http`, `server`, `location`	This is the timeout for sending data to the backend server. The timeout isn't applied to the entire response delay but rather between two write operations. Syntax: Time value (in seconds) Default value: `60`

`fastcgi_split_path_info`
 	
Context: `location`
	A directive particularly useful for URLs of the following form: `http://website.com/page.php/param1/param2/`
	
	The directive splits the path information according to the specified regular expression:
	
	`fastcgi_split_path_info ^(.+\.php)(.*)$;`
	
	This affects two variables:
	
	• `$fastcgi_script_name`: The filename of the actual script to be executed; for example; `page.php`
	• `$fastcgi_path_info`: The part of the URL that is after the script name; for example, `/param1/param2/`
	
	These can be employed in further parameter definitions:
	
	`fastcgi_param SCRIPT_FILENAME /home/website.com/www$fastcgi_script_name;`
	
	`fastcgi_param PATH_INFO $fastcgi_path_info;`
	
	Syntax: Regular expression

`fastcgi_store` Context: `http`, `server`, `location`	This directive enables a simple *cache store* where responses from the FastCGI application are stored as files on the storage device. When the same URI is requested again, the document is directly served from the cache store instead of forwarding the request to the FastCGI application. This directive enables or disables the cache store. Syntax: `on` or `off`

Directive	Description
`fastcgi_store_access` Context: `http`, `server`, `location`	This directive defines access permissions applied to the files created in the context of the cache store. Syntax: `fastcgi_store_access [user:r|w|rw] [group:r|w|rw] [all:r|w|rw];` Default: `fastcgi_store_access user:rw;`
`fastcgi_temp_path` Context: `http`, `server`, `location`	This directive sets the path of temporary and cache store files. Syntax: File path Example: `fastcgi_temp_path /tmp/nginx_fastcgi;`
`fastcgi_max_temp_file_size` Context: `http`, `server`, `location`	Set this directive to `0` to disable the use of temporary files for FastCGI requests or to specify a maximum file size. Default value: 1 GB Syntax: Size value Example: `fastcgi_max_temp_file_size 5m;`
`fastcgi_temp_file_write_size` Context: `http`, `server`, `location`	This directive sets the write buffer size when saving temporary files to the storage device. Syntax: Size value Default value: `2 * proxy_buffer_size`
`fastcgi_send_lowat` Context: `http`, `server`, `location`	This option allows you to make use of the `SO_SNDLOWAT` flag for TCP sockets under FreeBSD only. This value defines the minimum number of bytes in the buffer for output operations. Syntax: Numeric value (size) Default value: `0`

`fastcgi_pass_request_body`
 	
`fastcgi_pass_request_headers`
 	
Context: `http`, `server`, `location`
	These directives define whether or not, respectively, the request body and extra request headers should be passed on to the *backend* server.
	
	Syntax: `on` or `off`
	
	Default: `on`

Directive	Description
`fastcgi_ignore_headers` Context: `http`, `server`, `location`	This directive prevents NGINX from processing one or more of the following headers from the backend server response: • `X-Accel-Redirect` • `X-Accel-Expires` • `Expires` • `Cache-Control` • `X-Accel-Limit-Rate` • `X-Accel-Buffering` • `X-Accel-Charset` Syntax: `fastcgi_ignore_headers header1 [header2...];`

`fastcgi_next_upstream`
 	
Context: `http`, `server`, `location`
	When `fastcgi_pass` is connected to an upstream block, this directive defines cases where requests should be abandoned and resent to the next upstream server of the block. The directive accepts a combination of values among the following:
	
	• `error`: An error occurred while communicating or attempting to communicate with the server
	• `timeout`: A timeout occurred during transfers or connection attempts
	• `invalid_header`: The backend server returned an empty or invalid response
	• `http_500`, `http_503`, `http_403`, `http_404`, `http_429`: If such HTTP errors occur, NGINX switches to the next upstream server
	• `non-idempotent`: Allows retrying non-idempotent requests (`LOCK`, `POST`, `PATCH`)
	• `off`: Forbids using the next upstream server
	
	Examples:
	
	`fastcgi_next_upstream error timeout http_504;`
	
	`fastcgi_next_upstream timeout invalid_header;`

Directive	Description
`fastcgi_next_upstream_timeout` Context: `http`, `server`, `location`	Defines the timeout to be used in conjunction with `fastcgi_next_upstream`. Setting this directive to `0` disables it. Syntax: Time value (in seconds)
`fastcgi_next_upstream_tries` Context: `http`, `server`, `location`	Defines the maximum number of upstream servers tried before returning an error message, to be used in conjunction with `fastcgi_next_upstream`. Syntax: Numeric value (default: `0`)
`fastcgi_catch_stderr` Context: `http`, `server`, `location`	This directive allows you to intercept some of the error messages sent to `stderr` (**standard error** stream) and store them in the NGINX error log. Syntax: `fastcgi_catch_stderr filter;` Example: `fastcgi_catch_stderr "PHP Fatal error:";`
`fastcgi_keep_conn` Context: `http`, `server`, `location`	When set to `on`, NGINX will conserve the connection to the FastCGI server, thus reducing overhead. Syntax: `on` or `off` (default: `off`) Note that there is no equivalent directive in the uWSGI and SCGI modules.
`fastcgi_force_ranges` Context: `http`, `server`, `location`	When set to `on`, NGINX will enable byte-range support on responses from the FastCGI backend. Syntax: `on` or `off` (default: `off`)
`fastcgi_limit_rate` Context: `http`, `server`, `location`	Allows you to limit the rate at which NGINX downloads the response from the FastCGI backend. Syntax: Numeric value (bytes per second)

Table 5.2: A list of the main FastCGI directives

We've had a glance at the main FastCGI directives. Let's dig deeper into FastCGI cache and buffer directives.

FastCGI caching and buffering

Once you have correctly configured NGINX to work with your FastCGI application, you may optionally make use of `fastcgi_cache` directives, which will help you improve the overall server performance by setting up a cache system. Additionally, FastCGI buffering allows you to buffer responses from the

FastCGI backend instead of synchronously forwarding them to the client. You can get a full list of NGINX FastCGI cache modules on the official `https://nginx.org` website.

Here is a full NGINX FastCGI cache configuration example, making use of most of the cache-related directives described in the preceding table:

```
fastcgi_cache phpcache;
fastcgi_cache_key "$scheme$host$request_uri"; # $request_uri includes
the request arguments (such as /page.php?arg=value)
fastcgi_cache_min_uses 2; # after 2 hits, a request receives a cached
response
fastcgi_cache_path /tmp/cache levels=1:2 keys_zone=phpcache:10m
inactive=30m max_size=500M;
fastcgi_cache_use_stale updating timeout;
fastcgi_cache_valid 404 1m;
fastcgi_cache_valid 500 502 504 5m;
```

Since these directives are valid for pretty much any virtual host configuration, you may want to save these in a separate file (`fastcgi_cache`) that you include in the appropriate place:

```
server {
    server_name website.com;
    location ~* \.php$ {
         fastcgi_pass 127.0.0.1:9000;
         fastcgi_param SCRIPT_FILENAME /home/website.com/www$fastcgi_
script_name;
         fastcgi_param PATH_INFO $fastcgi_script_name;
         include fastcgi_params;
         include fastcgi_cache;
    }
}
```

We've covered most of the directives in the FastCGI module that you might find useful. Now, we are going to apply this knowledge to a real-life use case involving NGINX with PHP.

PHP with NGINX

We are now going to configure PHP to work together with NGINX via FastCGI. Why FastCGI in particular, as opposed to the other two alternatives, SCGI and uWSGI? The answer came with the release of PHP version `5.3.3`. As of this version, all releases come with an integrated FastCGI process manager, allowing you to easily connect applications implementing the FastCGI protocol. The only requirement is for your PHP build to have been configured with the `--enable-fpm` argument. If you are unsure whether your current setup includes the necessary components, worry not: a section of this chapter is dedicated to building PHP with everything we need. Alternatively, the `php-fpm` or `php8-fpm` packages can be found in most repositories.

Architecture

Before starting the setup process, it's important to understand the way PHP will interact with NGINX. We have established that FastCGI is a communication protocol running through sockets, which implies that there is a client and a server. The client is obviously NGINX. As for the server, well, the answer is actually more complicated than just "PHP."

By default, PHP supports the FastCGI protocol. The PHP binary processes scripts and is able to interact with NGINX via sockets. However, we are going to use an additional component to improve the overall process management—PHP-FPM:

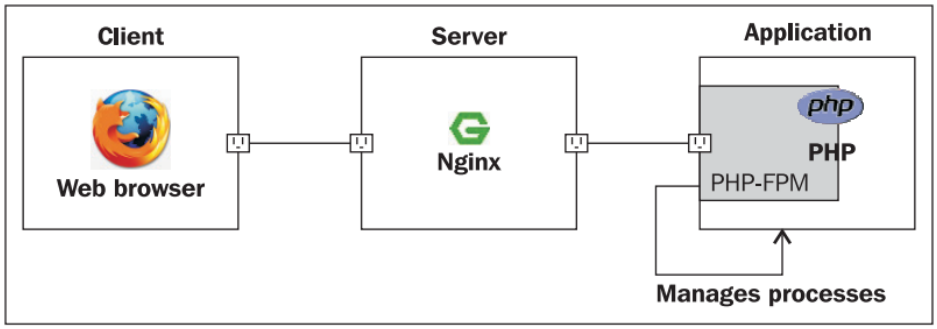


Figure 5.3: A diagram showing PHP-FPM running PHP in a sandbox

PHP-FPM takes FastCGI support to an entirely new level. Its numerous features are detailed in the next section.

PHP-FPM

The process manager, as its name suggests, is a script that manages PHP processes. It awaits and receives instructions from NGINX and runs the requested PHP scripts under the environment that you configure. In practice, PHP-FPM introduces a number of possibilities, such as the following:

- Automatically *daemonizing* PHP (turning it into a background process)
- Executing scripts in a *chrooted* (or sandboxed) environment
- Improved logging, IP address restrictions, pool separation, and much more

Setting up PHP and PHP-FPM

In this section, we will detail the process of downloading and installing PHP.

Installing using a package manager

At the time of writing these lines, the latest stable version of PHP is 8.3, but versions may differ depending on your distribution and the version of PHP included in the distribution.

For Red Hat-based systems and other systems using `dnf` as the package manager, execute the following command:

```
[root@local ~]# dnf install php-fpm
```

For Ubuntu, Debian, and other systems that use `apt`, execute the following command:

```
[root@local ~]# apt install php-fpm
```

Post-installation configuration

Begin by configuring your newly installed PHP; for example, by copying the `php.ini` file of your previous setup over the new one.

> **Note**
>
> Due to the way NGINX forwards script files and request information to PHP, a security breach might be caused by the use of the `cgi.fix_pathinfo=1` configuration option. It is highly recommended that you set this option to `0` in your `php.ini` file (it should be `0` by default on most distributions). For more information about this particular security issue, please consult the following article:
>
> `http://cnedelcu.blogspot.in/2010/05/nginx-php-via-fastcgi-important.html`.

The next step is to configure PHP-FPM. Open up the `php-fpm.conf` file. We cannot detail all aspects of the PHP-FPM configuration here (they are largely documented in the configuration file itself anyway), but there are important configuration directives that you shouldn't miss:

- Edit the user(s) and group(s) used by worker processes and, optionally, Unix sockets
- Address(es) and port(s) on which PHP-FPM will be listening
- Number of simultaneous requests that will be served
- IP address(es) allowed to connect to PHP-FPM

Running and controlling

Once you have made the appropriate changes to the PHP-FPM configuration file, you may start or restart it with the following command:

```
[root@local ~]# systemctl restart php-fpm
```

The preceding command will restart your `php-fpm` daemon; you can use `start` if your daemon hasn't been running yet.

> **Note**
>
> You can see the current `php-fpm` load and requests live by typing `systemctl status php-fpm`. This command can be very helpful as it will let you know whether your server is saturated or working properly.

NGINX configuration

If you have managed to configure and start PHP-FPM correctly, you are ready to tweak your NGINX configuration file to establish a connection between both parties. The following server block is a simple, valid template on which you can base your own website configuration:

```
server {
    server_name .website.com; # server name, accepting www
    listen 80; # listen on port 80
    root /home/website/www; # our root document path
    index index.php; # default request filename: index.php

    location ~* \.php$ { # for requests ending with .php
        # specify the listening address and port that you configured
previously
        fastcgi_pass 127.0.0.1:9000;
        # the document path to be passed to PHP-FPM
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_
name;
        # the script filename to be passed to PHP-FPM
        fastcgi_param PATH_INFO $fastcgi_script_name;
        # include other FastCGI related configuration settings
        include fastcgi_params;
    }
}
```

After saving the configuration file, reload NGINX:

```
[root@local ~]# systemctl reload nginx
```

Create a simple script at the root of your website to make sure PHP is being correctly interpreted:

```
[user@local ~]# echo "<?php phpinfo(); ?>" >/home/website/www/index.
php
```

Fire up your favorite web browser and load `http://localhost/` (or your website URL). You should see something similar to the following screenshot, which is the PHP server information page:

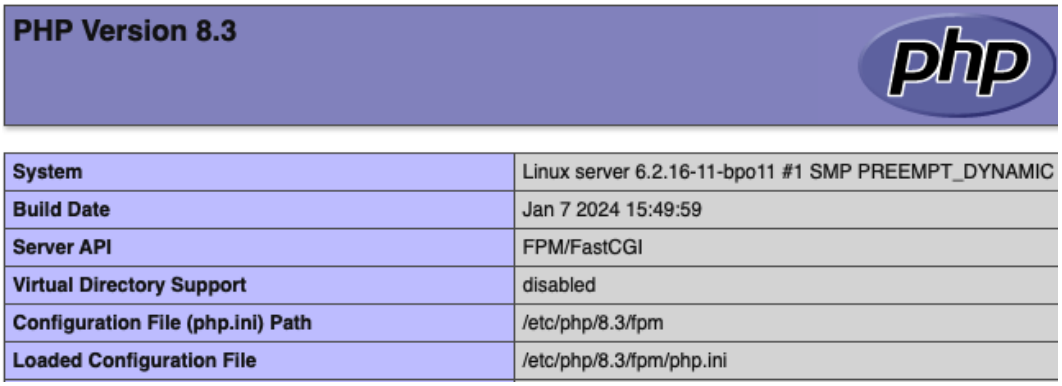


Figure 5.4: A screenshot of the PHP server information page

Note that you may run into the occasional `403 Forbidden` HTTP error if the file and directory access permissions are not properly configured. If that is the case, make sure that you specify the correct user and group in the `php-fpm.conf` file and that the directory and files are readable by PHP.

We've learned how to set up NGINX with PHP using the PHP-FPM server. In the next section, we'll learn how to do the same thing, this time with a Python server.

Python and NGINX

Python is a popular **object-oriented programming** (**OOP**) language available on many platforms, from Unix-based systems to Windows. It is also available for Java and the Microsoft .NET platform. If you are interested in configuring Python to work with NGINX, it's likely that you already have a clear idea of what Python does. We are going to use Python as a server-side web programming language, with the help of the Django framework.

Django

Django is an open source web development framework for Python that aims at making web development simple and easy, as its slogan states:

The web framework for perfectionists with deadlines.

More information is available on the project website at `https://www.djangoproject.com/`.

Among other interesting features, such as a dynamic administrative interface, a caching framework, and unit tests, Django comes with a FastCGI manager. Although the usual way to deploy Django is through WSGI, we're going to use Django's FastCGI manager as it's going to make things much simpler for us from the perspective of running Python scripts through NGINX.

Setting up Python and Django

We will now install Python and Django on your Linux operating system, along with its prerequisites. The process is relatively smooth and mostly consists of running a couple of commands that rarely cause trouble.

Python

Python should be available on your package manager repositories. To install it, run the following commands. For Red Hat-based systems and other systems using `dnf` as the package manager, use this command:

```
[root@local ~]# dnf install python python-devel
```

For Ubuntu, Debian, and other systems that use `apt`, use this command:

```
[root@local ~]# apt install python python-dev
```

The package manager will resolve dependencies by itself.

Django

In order to install Django, we will use a different approach (although you could skip this entirely and just install it from your usual repositories). We will be downloading the framework with `pip`, a tool that simplifies the installation of Python packages. Therefore, the first step is to install `pip`; for Red Hat-based systems and other systems using `dnf` as the package manager, execute the following command:

```
[root@local ~]# dnf install python-pip
```

For Ubuntu, Debian, and other systems that use `apt`, use this command:

```
[root@local ~]# apt install python-pip
```

The package manager will resolve dependencies by itself. Once `pip` is installed, run the following command to download and install Django 1.8.2, the latest stable version to date:

```
[root@website.com ~]# pip install Django==5.0.3
[...]
[root@website.com ~]# pip install -e django-trunk/
```

Finally, there is one last component required to run the Python FastCGI manager: the `flup` library. This provides the actual FastCGI protocol implementation. For Red Hat-based systems and other systems using `dnf` as the package manager (**Extra Packages for Enterprise Linux** (**EPEL**) repositories must be enabled; otherwise, you will need to build from source), use the following command:

```
[root@local ~]# dnf install python-flup
```

For Ubuntu, Debian, and other systems that use `apt`, use this command:

```
[root@local ~]# apt install python-flup
```

Starting the FastCGI process manager

The process of beginning to build a website with the Django framework is as simple as running the following command:

```
[root@website.com ~]# django-admin startproject mysite
```

Once that part is done, you will find a `manage.py` Python script that comes with the default project template. Open the newly created `mysite` directory containing `manage.py`, and run the following command:

```
[root@website.com mysite]# python manage.py runfcgi method=prefork
host=127.0.0.1 port=9000 pidfile=/var/run/django.pid
```

If everything is correctly configured and the dependencies are properly installed, running this command should produce no output, which is often a good sign. The FastCGI process manager is now running in the background, waiting for connections. You can verify that the application is running with the `ps` command (for example, by executing `ps aux | grep python`). If you don't see any running process, try changing the previous command slightly by selecting a different port. All we need to do now is to set up the virtual host in the NGINX configuration file.

NGINX configuration

The NGINX configuration is similar to the PHP one:

```
server {
    server_name .website.com;
listen 80;
# Insert the path of your Python project public files below
    root /home/website/www;
    index index.html;

    location / {
       fastcgi_pass 127.0.0.1:9000;
        fastcgi_param SCRIPT_FILENAME  $document_root$fastcgi_script_
name;
        fastcgi_param PATH_INFO $fastcgi_script_name;
        include fastcgi_params;
    }
}
```

We've now completed covering running NGINX with Python via FastCGI.

Summary

Whether you use PHP, Python, or any other CGI application, you should now have a clear idea of how to get your scripts processed behind NGINX. There are all sorts of implementations on the web for mainstream programming languages and the FastCGI protocol. Due to its efficiency, it is starting to replace server-integrated solutions such as Apache's `mod_php` and `mod_wsgi`, among others.

In the next chapter, we will learn how to use NGINX as a frontend for microservices, enabling us to improve the security and speed of web applications by running NGINX in front of them.

6

NGINX as a Reverse Proxy

The web has traditionally consisted of, relatively speaking, simple websites. The past few years have seen that change, though. The modern web comprises as many complex SaaS applications as it does personal blogs, news sites, and so on. As the web evolves, so does the list of technologies used to power these applications. No longer is it enough to just be a fast static file server with a FastCGI interface. These days, we need to consider technologies such as web sockets, as well as the expanded complexity of web application architectures and the demands they put on the front line of our web stack.

Thankfully, NGINX was originally built not only as a fast static file server but also as a reverse proxy. This means that NGINX was always intended to sit in front of other backend servers, farm out requests to different servers on the internal network, and serve up the response to the end user. In this chapter, we will take a look at the basics of how to do this with NGINX, and also at some of the more advanced things NGINX can do to make our life easier.

As such, we will cover the following main topics in the chapter:

- The reverse proxy mechanism
- The NGINX proxy module
- NGINX and microservices

Exploring the reverse proxy mechanism

Running NGINX as an application server is somewhat like the **FastCGI architecture** described in the previous chapter; we are going to be running NGINX as a frontend server, and for the most part, reverse proxy requests to our backend servers.

In other words, it will be in direct communication with the outside world whereas our backend servers, whether Node.js, Apache, and so on, will only exchange data with NGINX:

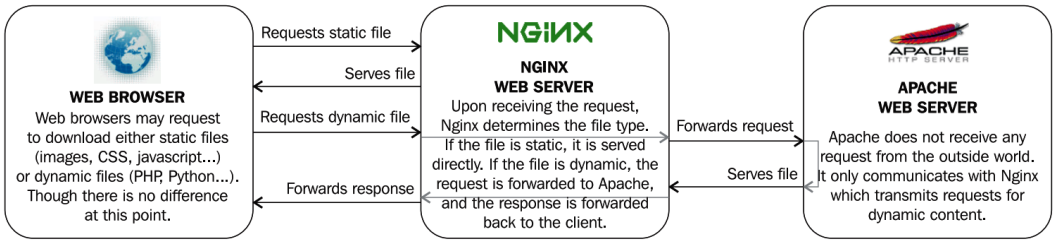


Figure 6.1: An example of using Nginx as a proxy server

There are now two web servers running and processing requests.

NGINX, positioned as a frontend server (in other words, as a reverse proxy), receives all the requests coming from the outside world. It filters them, either serving static files directly to the client or forwarding dynamic content requests to our backend server.

Our backend server only communicates with NGINX. It may be hosted on the same computer as the frontend, in which case the listening ports must be edited to leave ports `80` and `443` available to NGINX. Alternatively, you can employ multiple backend servers on different machines and load balance between them.

To communicate and interact with each other, neither process will be using FastCGI. Instead, as the name suggests, NGINX acts as a simple proxy server; it receives HTTP requests from the client (acting as the HTTP server) and forwards them to the backend server (acting as the HTTP client). There is thus no new protocol or software involved. The mechanism is handled by the proxy module of NGINX, as detailed later in this chapter.

Exploring the NGINX proxy module

Similar to the previous chapter, the first step toward establishing the new architecture will be to discover the appropriate module. The default NGINX build comes with the proxy module, which allows the forwarding of HTTP requests from the client to a backend server. We will be configuring multiple aspects of the module:

- Basic address and port information of the backend server
- Caching, buffering, and temporary file options
- Limits, timeout, and error behavior
- Other miscellaneous options

All these options are available via directives that we will learn to configure throughout this section.

Main directives

The first set of directives will allow you to establish a basic configuration such as the location of the backend server, information to be passed, and how it should be passed:

Directive	Description
`proxy_pass` Context: `location`, `if`	This specifies that the request should be forwarded to the backend server by indicating its location: • For regular HTTP forwarding, the syntax is `proxy_pass http://hostname:port;`. • For Unix domain sockets, the syntax is `proxy_pass http://unix:/path/to/file.socket;`. • You may also refer to upstream blocks `proxy_pass http://myblock;`. • Instead of `http://`, you can use `https://` for secure traffic. Additional URI parts as well as the use of variables are allowed. Examples: • `proxy_pass http://localhost:8080;` • `proxy_pass http://127.0.0.1:8080;` • `proxy_pass http://unix:/tmp/nginx.sock;` • `proxy_pass https://192.168.0.1;` • `proxy_pass http://localhost:8080/uri/;` • `proxy_pass http://unix:/tmp/nginx.sock:/uri/;` • `proxy_pass http://$server_name:8080;`

Directive	Description

`proxy_pass`
 	
Context: `location`, `if`
	Using an upstream block:
	
	`upstream backend {`
	
	`server 127.0.0.1:8080;`
	
	`server 127.0.0.1:8081;`
	
	`}`
	
	`location ~* .php$`
	
	`{`
	
	`proxy_pass http://backend;`
	
	`}`

`proxy_method` Context: `http`, `server`, `location`	This allows the overriding of the HTTP method of the request to be forwarded to the backend server. If you specify `POST`, for example, all requests forwarded to the backend server will be `POST` requests. Syntax: `proxy_method method;` Example: `proxy_method POST;`
`proxy_hide_header` Context: `http`, `server`, `location`	By default, as NGINX prepares the response received from the backend server to be forwarded back to the client, it ignores some of the headers, such as `Date`, `Server`, `X-Pad`, and `X-Accel-*`. With this directive, you can specify an additional header line to be hidden from the client. You may insert this directive multiple times with one header name for each. Syntax: `proxy_hide_header header_name;` Example: `proxy_hide_header Cache-Control;`
`proxy_pass_header` Context: `http`, `server`, `location`	Related to the previous directive, this directive forces some of the ignored headers to be passed on to the client. Syntax: `proxy_pass_header headername;` Example: `proxy_pass_header Date;`

Directive	Description

`proxy_pass_request_body`
 	
`proxy_pass_request_headers`
 	
Context: `http`, `server`, `location`
	This defines whether or not, respectively, the request body and extra request headers should be passed on to the backend server.
	
	Syntax: `on` or `off`
	
	Default: `on`

`proxy_redirect` Context: `http`, `server`, `location`	This allows you to rewrite the URL appearing in the Location HTTP header on redirections triggered by the backend server. Syntax: `off`, `default`, or the URL of your choice. `off`: Redirections are forwarded *as it is*. `default`: The value of the `proxy_pass` directive is used as the hostname and the current path of the document is appended. Note that the `proxy_redirect` directive must be inserted after the `proxy_pass` directive as the configuration is parsed sequentially. URL: Replace a part of the URL with another. Additionally, you may use variables in the rewritten URL. Examples: • `proxy_redirect off;` • `proxy_redirect default;` • `proxy_redirect http://localhost:8080/ http://example.com/;` • `proxy_redirect http://localhost:8080/wiki/ /w/;` • `proxy_redirect http://localhost:8080/ http://$host/;`

Directive	Description
`proxy_next_upstream` Context: `http`, `server`, `location`	When `proxy_pass` is connected to an upstream block, this directive defines the cases where requests should be abandoned and resent to the next upstream server of the block. The directive accepts a combination of values among the following: • `error`: An error occurred while communicating or attempting to communicate with the server • `timeout`: A timeout occurred during transfers or connection attempts • `invalid_header`: The backend server returned an empty or invalid response • `http_500`, `http_502`, `http_503`, `http_504`, `http_40`: In case such HTTP errors occur, NGINX switches to the next upstream • `off`: Forbids you from using the next upstream server Examples: • `proxy_next_upstream error timeout http_504;` • `proxy_next_upstream timeout invalid_header;`
`proxy_next_upstream_timeout` Context: `http`, `server`, `location`	This defines the timeout to be used in conjunction with `proxy_next_upstream`. Setting this directive to `0` disables it. Syntax: Time value (in seconds)
`proxy_next_upstream_tries` Context: `http`, `server`, `location`	This defines the maximum number of upstream servers tried before returning an error message, to be used in conjunction with `proxy_next_upstream`. Syntax: Numeric value (default: `0`)

Table 6.1: The main directives for the proxy module

Caching, buffering, and temporary files

Ideally, as much as possible, you should reduce the number of requests being forwarded to the backend server. The following directives will help you build a caching system, as well as control buffering options and the way NGINX handles temporary files:

Directive	Description
`proxy_buffer_size` Context: `http`, `server`, `location`	This sets the size of the buffer for reading the beginning of the response from the backend server, which usually contains simple header data. The default value corresponds to the size of `1` buffer, as defined by the previous directive (`proxy_buffers`). Syntax: Numeric value (size) Example: `proxy_buffer_size 4k;`
`proxy_buffering, proxy_request_buffering` Context: `http`, `server`, `location`	This defines whether or not the response from the backend server should be buffered (or client requests, in the case of `proxy_request_buffering`). If set to `on`, NGINX will store the response data in memory using the memory space offered by the buffers. If the buffers are full, the response data will be stored as a temporary file. If the directive is set to `off`, the response is directly forwarded to the client. Syntax: `on` or `off` Default: `on`
`proxy_buffers` Context: `http`, `server`, `location`	This sets the amount and size of buffers that will be used for reading the response data from the backend server. Syntax: `proxy_buffers amount size;` Default: 8 buffers, `4k` or `8k` each, depending on the platform Example: `fastcgi_buffers 8 4k;`
`proxy_busy_buffers_size` Context: `http`, `server`, `location`	When the backend-received data accumulated in buffers exceeds the specified value, buffers are flushed and data is sent to the client. Syntax: Numeric value (size) Default: `2 * proxy_buffer_size`
`proxy_cache` Context: `http`, `server`, `location`	This defines a cache zone. The identifier given to the zone is to be reused in further directives. Syntax: `proxy_cache zonename;` Example: `proxy_cache cache1;`

Directive	Description
`proxy_cache_key` Context: `http`, `server`, `location`	This directive defines the cache key; in other words, it differentiates one cache entry from another. If the cache key is set to `$uri`, as a result, all requests with `$uri` will work as a single cache entry. But that's not enough for most dynamic websites. You also need to include the query string arguments in the cache key so that `/index.php` and `/index.php?page=contact` do not point to the same cache entry. Syntax: `proxy_cache_key key;` Example: `proxy_cache_key "$scheme$host$request_uri $cookie_user";`
`proxy_cache_path` Context: `http`	This indicates the directory for storing cached files, as well as other parameters. Syntax: `proxy_cache_path path [use_temp_path=on|off] [levels=numbers keys_zone=name:size inactive=time max_size=size];` The additional parameters are as follows: • `use_temp_path`: Set this flag to `on` if you want to use the path defined via the `proxy_temp_path` directive • `levels`: This indicates the depth level of subdirectories (usually *1:2* is enough) • `keys_zone`: This lets you make use of the zone you previously declared with the `proxy_cache` directive and indicates the size to occupy in memory • `inactive`: If a cached response is not used within the specified timeframe, it is removed from the cache • `max_size`: This defines the maximum size of the entire cache Example: `proxy_cache_path /tmp/nginx_cache levels=1:2 zone=zone1:10m inactive=10m max_size=200M;`

Directive	Description
`proxy_cache_methods` Context: `http`, `server`, `location`	This defines the HTTP methods eligible for caching. `GET` and `HEAD` are included by default and cannot be disabled. Syntax: `proxy_cache_methods METHOD;` Example: `proxy_cache_methods OPTIONS;`
`proxy_cache_min_uses` Context: `http`, `server`, `location`	This defines the minimum number of hits before a request is eligible for caching. By default, the response of a request is cached after one hit (next requests with the same cache key will receive the cached response). Syntax: Numeric value Example: `proxy_cache_min_uses 1;`
`proxy_cache_valid` Context: `http`, `server`, `location`	This directive allows you to customize the caching time for different kinds of response codes. You may cache responses associated with `404` error codes for `1` minute, and on the opposite cache, `200 OK` responses for `10` minutes or more. This directive can be inserted more than once: • `proxy_cache_valid 404 1m;` • `proxy_cache_valid 500 502 504 5m;` • `proxy_cache_valid 200 10;` Syntax: `proxy_cache_valid code1 [code2...] time;`
`proxy_cache_use_stale` Context: `http`, `server`, `location`	This defines whether or not NGINX should serve stale cached data in certain circumstances (in regard to the gateway). If you use `proxy_cache_use_stale timeout`, and if the gateway times out, then NGINX will serve cached data. Syntax: `proxy_cache_use_stale [updating] [error] [timeout] [invalid_header] [http_500];` Example: `proxy_cache_use_stale error timeout;`

Directive	Description
`proxy_max_temp_file_size` Context: `http`, `server`, `location`	Set this directive to `0` to disable the use of temporary files for requests eligible for proxy forwarding or specify a maximum file size. Syntax: Size value Default value: 1 GB Example: `proxy_max_temp_file_size 5m;`
`proxy_temp_file_write_size` Context: `http`, `server`, `location`	This sets the write buffer size when saving temporary files to the storage device. Syntax: Size value Default value: `2 * proxy_buffer_size`
`proxy_temp_path` Context: `http`, `server`, `location`	This sets the path of temporary and cache store files. Syntax: `proxy_temp_path path [level1 [level2...]]` Examples: • `proxy_temp_path /tmp/nginx_proxy;` • `proxy_temp_path /tmp/cache 1 2;`

Table 6.2: The caching, buffering, and temporary file directives for the proxy module

Limits, timeouts, and errors

The following directives will help you define timeout behavior, as well as various limitations regarding communications with the backend server:

Directive	Description
`proxy_connect_timeout` Context: `http`, `server`, `location`	This defines the backend server connection timeout. This is different from the read/send timeout. If NGINX is already connected to the backend server, the `proxy_connect_timeout` is not applicable. Syntax: `Time value` (in seconds) Example: `proxy_connect_timeout 15;`

`proxy_read_timeout` Context: `http`, `server`, `location`	This is the timeout for reading data from the backend server. This timeout isn't applied to the entire response delay but between two read operations instead. Syntax: `Time value` (in seconds) Default value: `60` Example: `proxy_read_timeout 60;`
`proxy_send_timeout` Context: `http`, `server`, `location`	This timeout is for sending data to the backend server. The timeout isn't applied to the entire response delay but between two write operations instead. Syntax: `Time value` (in seconds) Default value: `60` Example: `proxy_send_timeout 60;`
`proxy_ignore_client_abort` Context: `http`, `server`, `location`	If set to `on`, NGINX will continue processing the proxy request, even if the client aborts its request. In the other case (`off`), when the client aborts its request, NGINX also aborts its request to the backend server. Default value: `off`
`proxy_intercept_errors` Context: `http`, `server`, `location`	By default, NGINX returns all error pages (HTTP status code `400` and higher) sent by the backend server directly to the client. If you set this directive to `on`, the error code is parsed and can be matched against the values specified in the `error_page` directive. Default value: `off`
`proxy_send_lowat` Context: `http`, `server`, `location`	This option allows you to make use of the `SO_SNDLOWAT` flag for TCP sockets under BSD-based operating systems only. This value defines the minimum number of bytes in the buffer for output operations. Syntax: Numeric value (size) Default value: `0`
`proxy_limit_rate` Context: `http`, `server`, `location`	This allows you to limit the rate at which NGINX downloads the response from the backend proxy. Syntax: Numeric value (bytes per second)

Table 6.3: The directives relevant to limits, timeouts, and errors

Other directives

Finally, the last set of directives available in the proxy module is uncategorized and is as follows:

Directive	Description
`proxy_headers_hash_max_size` Context: `http`, `server`, `location`	This sets the maximum size for the proxy header's hash tables. Syntax: Numeric value Default value: `512`
`proxy_headers_hash_bucket_size` Context: `http`, `server`, `location`	This sets the bucket size for the proxy header's hash tables. Syntax: Numeric value Default value: `64`
`proxy_force_ranges` Context: `http`, `server`, `location`	When set to `on`, NGINX will enable byte-range support on responses from the backend proxy. Syntax: `on` or `off` Default value: `off`
`proxy_ignore_headers` Context: `http`, `server`, `location`	This prevents NGINX from processing one of the following four headers from the backend server response: `X-Accel-Redirect`, `X-Accel-Expires`, `Expires`, and `Cache-Control`. Syntax: `proxy_ignore_headers header1 [header2...];`
`proxy_set_body` Context: `http`, `server`, `location`	This allows you to set a static request body for debugging purposes. Variables may be used in the directive value. Syntax: String value (any value) Example: `proxy_set_body test;`
`proxy_set_header` Context: `http`, `server`, `location`	This directive allows you to redefine header values to be transferred to the backend server. It can be declared multiple times. Syntax: `proxy_set_header Header Value;` Example: `proxy_set_header Host $host;`

Directive	Description
`proxy_store` Context: `http`, `server`, `location`	This specifies whether or not the backend server response should be stored as a file. Stored response files can be reused for serving other requests. Possible values are `on`, `off`, or a path relative to the document root (or alias). You may also set this to `on` and define the `proxy_temp_path` directive. Examples: • `proxy_store on;` • `proxy_temp_path /temp/store;`
`proxy_store_access` Context: `http`, `server`, `location`	This directive defines file access permissions for the stored response files. Syntax: `proxy_store_access [user:[r|w|rw]] [group:[r|w|rw]] [all:[r|w|rw]];` Example: `proxy_store_access user:rw group:rw all:r;`
`proxy_http_version` Context: `http`, `server`, `location`	This sets the HTTP version to be used for communicating with the proxy backend. HTTP `1.0` is the default value, but if you are going to enable keepalive connections, you might want to set this directive to `1.1`. Syntax: `proxy_http_version 1.0 | 1.1;`

`proxy_cookie_domain`
 	
`proxy_cookie_path`
 	
Context: `http`, `server`, `location`
	This applies an on-the-fly modification to the domain or path attributes of a cookie (case-insensitive).
	
	Syntaxes:
	
	• `proxy_cookie_domain off | domain replacement;`
	• `proxy_cookie_path off | domain replacement ;`

Table 6.4: Uncategorized directives for the proxy module

Variables

The proxy module offers several variables that can be inserted in various locations, for example, in the `proxy_set_header` directive or the logging-related directives such as `log_format`. The available variables are the following:

- `$proxy_host`: This contains the hostname of the backend server used for the current request.
- `$proxy_port`: This contains the port of the backend server used for the current request.
- `$proxy_add_x_forwarded_for`: This variable contains the value of the `X-Forwarded-For` request header followed by the remote address of the client. Both values are separated by a comma. If the `X-Forwarded-For` request header is unavailable, the variable only contains the client's remote address.
- `$proxy_internal_body_length`: Length of the request body (set with the `proxy_set_body` directive or `0`).

Although the directives and variables mentioned are for informational purposes at this stage, we will actively apply them in upcoming chapters, exploring their practical uses in scenarios such as load balancing and beyond.

Looking at NGINX and microservices

Now that we've explored the proxy module in depth, it's time to have a look at what a modern web application architecture might look like. There are entire books dedicated to this topic but we only really need to know how NGINX can enable various setups, and the NGINX part doesn't differ too much between different setups.

For any given task that we need our application to do, we have two options: we can either proxy to a backend server such as Node.js and have that handle the work, or we can implement it directly in NGINX. Which option you go with depends on a lot of factors, but the two main factors to consider are speed and complexity.

Proxying to a complex backend server has an overhead cost but usually allows you to code reusability and use package managers such as Packagist and NPM. Conversely, implementing a feature in NGINX puts us closer to the user so we have less overhead, but the development itself also becomes more difficult.

Most setups will choose to proxy to a backend for simplicity. An example of a feature implemented in NGINX would be Cloudflare and its proxy/CDN service. Since they deal with a huge scale of requests and response time is critical to them, they have implemented their security filtering (web application firewall) directly in NGINX using a module to add Lua support in the NGINX config file.

Cloudflare has hundreds of developers, including people who have worked on the core part of NGINX code before, so don't expect to quite reach their level, but there are also simpler scenarios where NGINX can implement part of the application logic.

A simple example of application logic in NGINX is to move our cache from inside our backend server to NGINX itself. In the following example, we're checking Memcached for a cached version of a page, and only if we don't find it do we proxy to our application backend:

```
# Check cache and use PHP as a fallback.
location ~* \.php$ {
default_type text/html;
charset utf-8;

if ($request_method = GET) {
set $memcached_key $request_uri;

memcached_pass memcached;
error_page 404 502 = @nocache;
}

if ($request_method != GET) {
fastcgi_pass backend;
}
}

location @nocache {
fastcgi_pass backend;
}
```

When we go into more advanced logic, the NGINX configuration gets a bit complicated. We will look at several proxy configuration scenarios in more detail in the next chapter.

Summary

In this chapter, we had a look at how reverse proxying works and how NGINX fits into the modern picture of microservices and complex web applications, both in the sense of enabling the microservice architecture and also in the sense of building application logic directly into NGINX.

This chapter should have given you an idea of the possibilities that NGINX provides as an application server, and hopefully clarified the complexity/speed trade-off of implementing logic in NGINX.

Now that we've got an overview of the possibilities offered by NGINX, we're going to put what we've learned to good use. In the next chapter, we'll cover a specific case: NGINX with Docker, using the NGINX proxy.

Part 3: NGINX in Action

In this final part of the book, you'll explore how NGINX can be integrated into larger IT infrastructures, with advanced deployment strategies, cloud environments, and automated management. This section focuses on the real-life use cases of NGINX within complex systems, including load balancing, cloud deployments, and maintaining high availability and security.

This part includes the following chapters:

- *Chapter 7, Introduction to Load Balancing and Optimization*
- *Chapter 8, NGINX within a Cloud Infrastructure*
- *Chapter 9, Fully Deploy, Manage, and Auto-Update NGINX with Ansible*
- *Chapter 10, Case Studies*
- *Chapter 11, Troubleshooting*

7

Introduction to Load Balancing and Optimization

As much as NGINX will help your servers hold the load, there are always limits to what a single machine can process; an aging hard drive or limited bandwidth will eventually induce a bottleneck, resulting in longer request-serving times, which, in turn, leads to the disappointment of your visitors.

As your websites grow more popular and your single machine begins to suffer, you will be tempted to simply get a bigger and more expensive server. But this would not be a cost-efficient approach in the long run, and remember that the more strain a server is exposed to, the more likely it is to suffer from hardware failure.

In this chapter, we will investigate two concepts, the first of which is load balancing: the art of distributing a load across several servers and managing this distribution efficiently. The second part will explore the subject of thread pools: a new mechanism relieving servers under heavy loads (more specifically, loads induced by blocking operations) by serving requests in a slightly different manner.

This chapter covers the following topics:

- Introducing load balancing
- Using NGINX as a TCP load balancer
- Exploring thread pools and I/O mechanisms

Introducing load balancing

All of the most visited websites in the world are built over carefully planned server architectures; fast page loads and download speeds are a requirement for long-term traffic growth. The concept of **load balancing** has the potential to solve problems pertaining to scalability, availability, and performance. After a quick description of the concept, we will elaborate on how NGINX offers to implement such an architecture.

Understanding the concept of load balancing

To put it simply, the concept of load balancing consists of distributing the workload (CPU load, hard disk load, or other forms) across several servers, in a manner that is completely transparent to your visitors.

In the case of a single-server architecture, client requests are received and processed by one machine. A machine has a limited capacity of operation; for example, a web server that is able to respond to 1,000 HTTP requests per second. If the server receives more than 1,000 requests per second, the 1,001st client request received in that second will not be served in a timely manner. From then on, page-serving speeds would begin to increase, resulting in a degraded experience for its visitors:

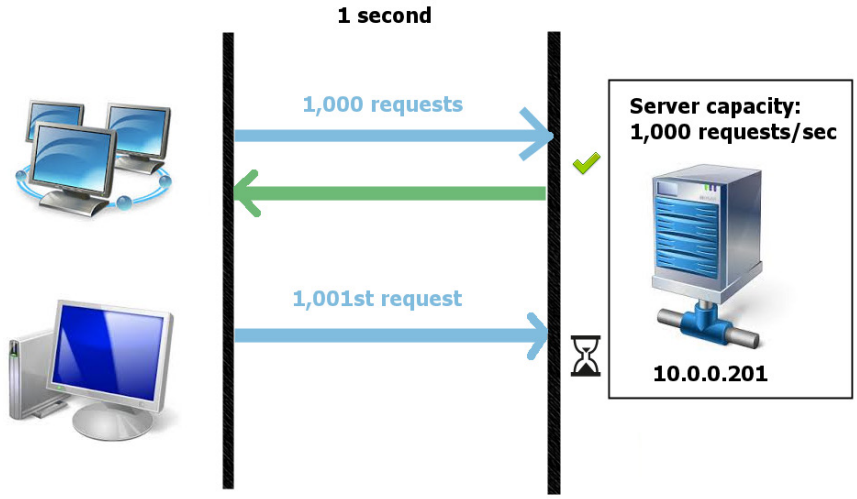


Figure 7.1: An example of how request tops are managed

Distributing a load across several servers increases the overall request-serving capacity; with two servers at your disposal, you could theoretically allow 2,000 HTTP requests to be served per second. With three servers, you could serve 3,000 requests, and so on.

There are several techniques available for achieving load balancing, with **DNS load balancing** being one of the most commonly implemented techniques. When a person wishes to visit your website, their web browser will resolve your domain name (`example.com`) into an IP address (`1.2.3.4`). To achieve DNS load balancing, simply associate multiple IP addresses to your domain. Upon visiting your website, the operating systems of your visitors will select one of these IP addresses following a **simple round-robin** algorithm, thus ensuring that on a global scale, all of your servers receive more or less the same amount of traffic:

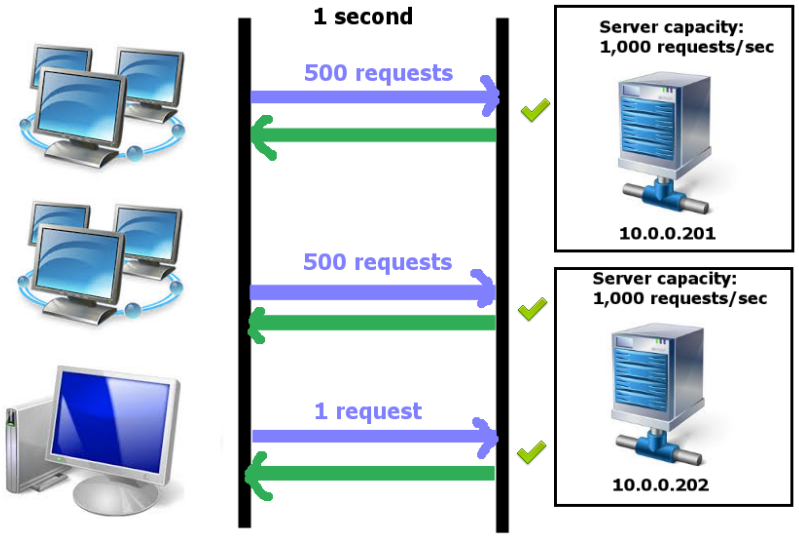


Figure 7.2: Another example of how request tops are managed

Albeit simple to implement, this load-balancing method cannot always be applied to high-traffic websites because it has several major issues:

- What if the IP address selected by a visitor's operating system points to a server that is temporarily unavailable?
- What if your architecture is made of several types of servers, some of which are capable of handling more requests than others?
- What if a visitor connects to a particular server and logs in to their user account, only to get switched to another server 10 minutes later, losing their session data?

The last of these issues is also known as the **session affinity** problem and is further detailed in the next section.

Session affinity

Session affinity is an expression that designates the persistent assignment of a client to a particular server in a load-balanced infrastructure. We use the word *session* to describe a set of requests performed by a client to a server. When a visitor browses a website, they often visit more than one page: they log in to their account, they add a product to their shopping cart, they check out, and so on. Until they close their web browser (or a tab), all of their subsequent page views are part of a session, which is most of the time stateful: the server conserves data relative to the operations performed during the visit. In our example, that server would remember the contents of the shopping cart and the login credentials.

If, at some point during the session, the visitor were to switch servers and connect to **Server B**, they would lose any session information contained on **Server A**. The visitor would then lose the contents of their shopping cart, as well as their login credentials (they would get logged out):

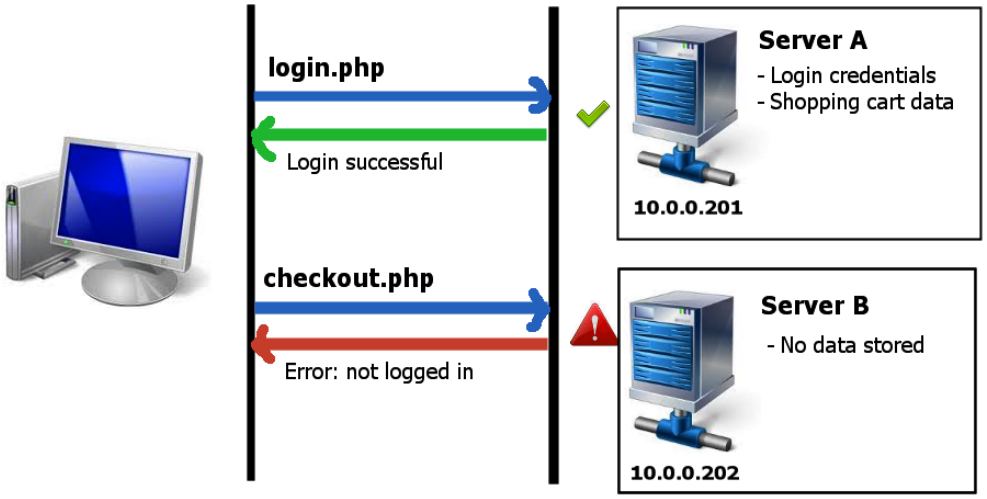


Figure 7.3: When switching the backend server, we might lose the existing session

For that reason, it is of utmost importance to maintain session affinity: in other words, to ensure that a visitor remains assigned to a particular server at all times. The DNS load-balancing method does not ensure session affinity, but fortunately, NGINX will help you achieve it.

The upstream module

The implementation of load balancing in NGINX is particularly clever as it allows you to distribute a load at several levels of your infrastructure. It isn't limited to proxying HTTP requests across backend servers; it also offers to distribute requests across FastCGI backends (FastCGI, uWSGI, SCGI, and more), or even distribute queries to Memcached servers. Any directive that ends with `_pass`, such as `proxy_pass`, `fastcgi_pass`, or `memcached_pass`, accepts a reference to a group of servers.

The first step is to declare this group of servers with the help of the *upstream* block, which must be placed within the `http` block. Within the `upstream` block, declare one or more servers with the `server` directive:

```
http {
    upstream MyUpstream {
        server 10.0.0.201;
        server 10.0.0.202;
        server 10.0.0.203;
}
```

```
[...]
}
```

Alternatively, you can also use `include` inside your upstream block to load servers from an external file:

```
http {
    upstream MyUpstream {
        include myUpstreamServers.txt
}
[...]
}
```

Now that your server group is declared, you can reference it in your virtual host configuration. For example, you can distribute incoming HTTP requests across the server group simply by proxying them:

```
server {
    server_name example.com;
    listen 80;
    root /home/example.com/www;
# Proxy all requests to the MyUpstream server group
proxy_pass http://MyUpstream;
    [...]
}
```

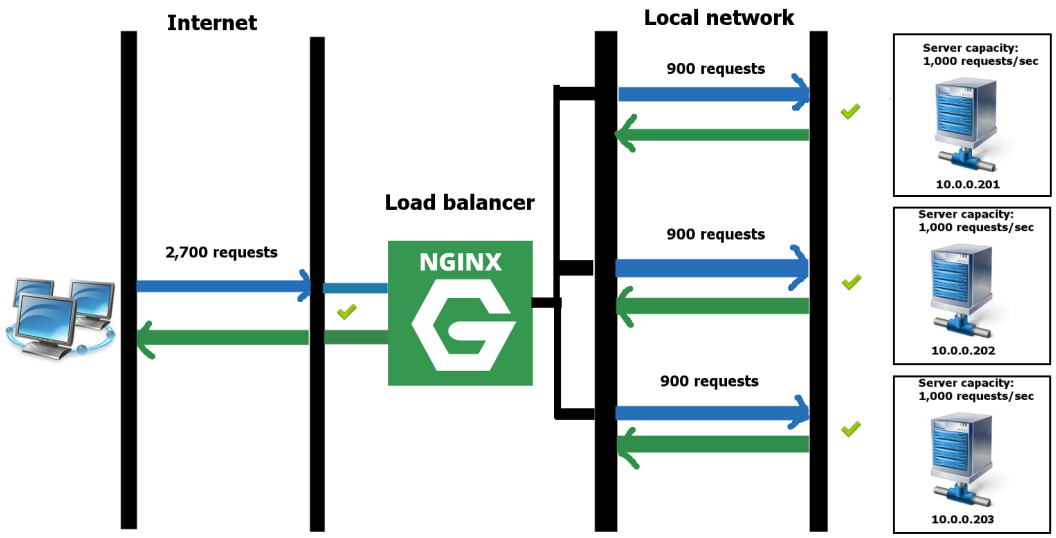


Figure 7.4: An example of Nginx acting as a relay for internal servers

In this most basic state of configuration, requests are distributed across the three servers of the `MyUpstream` group according to a simple round-robin algorithm, without maintaining session affinity.

Request distribution mechanisms

NGINX offers several ways to solve the problems we mentioned earlier. The first and simplest of them is the `weight` flag, which can be enabled in the definition of your server group:

```
upstream MyUpstream {
    server 10.0.0.201 weight=3;
    server 10.0.0.202 weight=2;
    server 10.0.0.203;
}
```

By default, servers have a weight of `1`, unless you specify otherwise. Such a configuration enables you to give more importance to particular servers; the higher their weight, the more requests they will receive from NGINX. In this example, for every six HTTP requests received, NGINX will systematically distribute the following:

- Three requests to the `10.0.0.201` server (`weight=3`)
- Two requests to the `10.0.0.202` server (`weight=2`)
- One request to the `10.0.0.203` server (`weight=1`)

For every 12 requests, NGINX will distribute the following:

- Six requests to the `10.0.0.201` server (`weight=3`)
- Four requests to the `10.0.0.202` server (`weight=2`)
- Two requests to the `10.0.0.203` server (`weight=1`)

NGINX also includes a mechanism that will verify the state of servers in a group. If a server doesn't respond in time, the request will be re-sent to the next server in the group. There are several flags that can be assigned to servers in an upstream block that will allow you to better control this mechanism:

- `fail_timeout=N`, where `N` is the number of seconds before a request is considered to have failed.
- `max_fails=N`, where `N` is the number of attempts that should be performed on a server before NGINX gives up and switches to the next server. By default, NGINX only tries once. If all servers become unresponsive, NGINX will wait for `fail_timeout` to expire before resetting all server fail counts and trying again.
- `max_conns=N`, where `N` is the number of maximum concurrent connections that can be sent to that server. By default, NGINX will not limit concurrent connections.
- `backup` marks the server as a backup server, instructing NGINX to use it only in the case of failure of another server (it is not used otherwise).
- `down` marks the server as permanently unavailable, instructing NGINX not to use it anymore.

Finally, NGINX offers plenty of options to achieve session affinity. They come in the form of directives that should be inserted within the upstream block. The simplest of them is `ip_hash`; this directive instructs NGINX to calculate a hash from the first 3 bytes of the client IPv4 address (or the full IPv6 address) and, based on that hash, keep the client assigned to a particular server. As long as the client IP address remains the same, NGINX will always forward requests to the same server in the upstream group:

```
upstream {
    server 10.0.0.201 weight=3;
    server 10.0.0.202 weight=2;
    server 10.0.0.203;
    ip_hash;
}
```

Some administrators may deem this method too unreliable, considering the fact that a majority of internet service providers across the globe still provide dynamic IP addresses, renewed on a 24-hour basis. So why not use your own distribution key? Instead of the client IP address, you could separate requests based on the criteria of your choice, thanks to the `hash` directive. Since the directive allows variables, you could decide to separate requests based on a cookie value:

```
upstream {
    server 10.0.0.201;
    server 10.0.0.202;
    hash $cookie_username;
}
```

Based on the data contained in the `username` cookie, your visitors will be assigned to the first or the second server in the upstream group.

We have just seen how to use NGINX as an HTTP load balancer. In the next section, we'll look at how to get the NGINX load balancer working, but this time using TCP instead of HTTP.

Using NGINX as a TCP/UDP load balancer

Until recently, the open source version of NGINX would only allow load balancing in the context of HTTP requests. In the meantime, the commercial subscription NGINX Plus took the concept one step further: using NGINX as a TCP/UDP load balancer. This would pave the way to much broader possibilities; you could then set up NGINX to distribute the load across any form of networked servers—database servers, email servers, literally everything that communicates via TCP. In May 2015, the authors decided that TCP/UDP load balancing should be part of the open source version. As of NGINX 1.9.0, the stream module is included in the source code readily available at `https://nginx.org/`.

The stream module

The way TCP/UDP load balancing works in NGINX is remarkably similar to HTTP load balancing. However, since the module that brings forth the new set of directives is not included in the default build, you will need to run the `configure` command with the following flag before building the program:

```
--with-stream
```

The stream module offers a new block called **stream**, which must be placed at the root of the configuration file (outside of the `http` block). In this block, you must declare two sets of directives:

- `server` declares a TCP/UDP server listening on a particular port, and optionally, a network interface, with or without SSL
- `upstream` defines a server group in a similar manner as seen previously

In your server blocks, the requests will be sent to the server group with the `proxy_pass` directive.

An example of MySQL load balancing

If you already understand how HTTP load balancing works in NGINX, the following example will look spectacularly simple to you. We will configure NGINX to receive MySQL connections and balance them across two backend servers:

```
stream {
  upstream MyGroup {
    # use IP address-based distribution
    hash $remote_addr;
    server 10.0.0.201 weight=2;
    server 10.0.0.202;
    server 10.0.0.203 backup; # use as backup only
  }
  server {
    # listen on the default MySQL port
    listen 3306;
    proxy_pass MyGroup; # forward requests to upstream
  }
}
```

That's all there is to it. All directives and options offered by the upstream module are still there, but keep in mind that you won't be able to use HTTP-based variables (such as cookies) to achieve session affinity. The stream module comes with a lot more options and flags, but they are not detailed here, as this falls outside the scope of an HTTP server; additional documentation can be found at `https://nginx.org/`.

We now have an overview of how to run a load balancer using NGINX for both HTTP and TCP/UDP requests. In the next section, we'll look at threads and I/O to better understand and improve your server's resource management under heavy loads.

Exploring thread pools and I/O mechanisms

Before making important financial decisions, such as investing in an additional server or two, you should look to optimize your current setup to make the most of your existing infrastructure.

Relieving worker processes

In the case of websites that require heavy I/O operations, such as file uploads or downloads, the asynchronous architecture of NGINX can present a certain disadvantage: while the master process is able to absorb incoming connections asynchronously, worker processes can be blocked for relatively long periods of time by certain tasks (the most common of which is reading data from hard disk drives or network drives).

Consider a simplified configuration with two worker processes; each HTTP request received by NGINX gets assigned to either process. Within a process, operations are performed sequentially: receiving and parsing the request, reading the requested file from its storage location, and finally, preparing and sending the response to the client. If for some reason you were to serve files stored on a network drive with a latency of about 100 ms, both of your worker processes would be spending most of their time waiting for the files. As a result, your server would only be able to serve 18 to 20 requests per second:

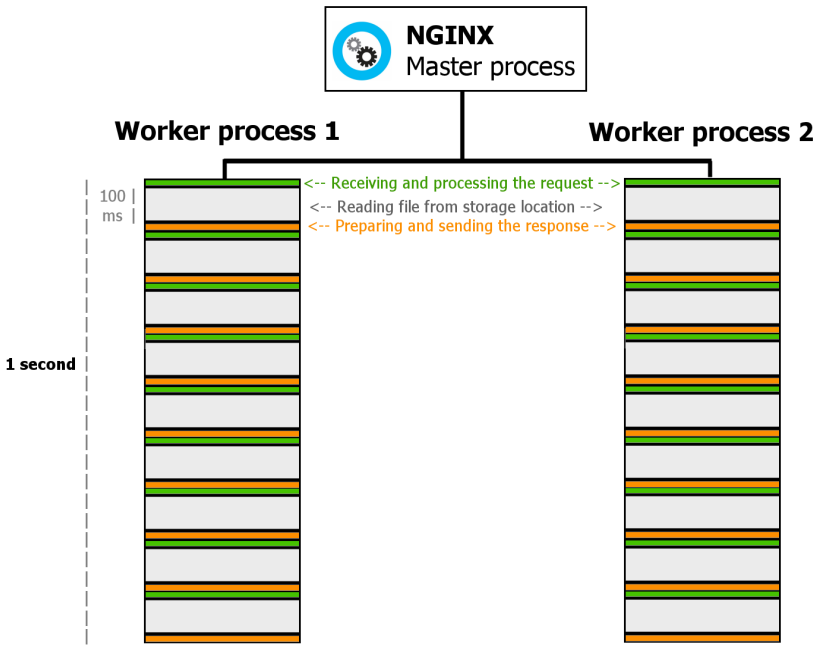


Figure 7.5: An explanation of how worker processes and latency work

This isn't just a problem that occurs for network drives. Even regular hard disk drives can take a certain time to fetch a file if it isn't in the cache; a 10 ms latency isn't insignificant when you multiply it by 1,000!

The solution that has been made available as of NGINX 1.7.11 is called **thread pools**. The basic principle behind this solution is that instead of reading files synchronously within the worker process, NGINX delegates the operation to a thread. This immediately liberates the worker process, which can then move on to the next request in the queue. Whenever the thread finishes performing the operation, the worker process finalizes and sends the response to the client. It is a pretty simple concept to understand, and thankfully, it's just as simple to configure.

AIO, Sendfile, and DirectIO

In order to enable support for thread pools, NGINX must be built with the `--with-threads` parameter; this functionality doesn't come by default. The first step of the configuration is to define a thread pool with the `thread_pool` directive at the root of your configuration file.

Syntax: `thread_pool name threads=N [max_queue=Q];`

In this syntax, `name` is the name you wish to give to the thread pool, `N` is the number of threads that should be spawned, and `Q` is the maximum number of operations allowed in the queue. By default, a thread pool exists with the name `default`, coming with 32 threads and a maximum queue of 65,536 operations.

In `location` blocks that require it, simply insert the `aio` directive and specify the thread pool name:

```
thread_pool MyPool threads=64;
[...]
location /downloads/ {
    aio threads=MyPool;
}
```

Alternatively, insert `aio threads` without a pool name if you want to use the default thread pool. It is also possible to use both `sendfile` and `aio` in the same location:

```
location /downloads/ {
aio threads;
directio 8k;
sendfile on;
}
```

If the file requested by the client is over `8k` (the value specified with the `directio` directive), `aio` will be used. Otherwise, the file will be sent via `sendfile`. For a deeper dive into the specifics of `sendfile` and `directio`, we encourage you to consult the official NGINX documentation.

We now have a better understanding of how to manage NGINX server resources, thanks in particular to thread pools. Now it's time to summarize what we've learned in this chapter.

Summary

Before adapting your infrastructure to increasingly high traffic, you should always look for solutions offered by your current set of tools. If traffic causes your server to become unresponsive because of blocking operations, such as slow disk reads, you should give thread pools a try. If this turns out to be insufficient, load balancing is the next best thing. Thankfully, as we have discovered in this chapter, implementing a load-balanced architecture is made particularly easy by NGINX; you can even use it to distribute the load of other server applications such as MySQL, email, and more.

Now that we have seen a basic yet comprehensive approach to the most advanced mechanisms offered by NGINX, let's move on to deploying NGINX in a cloud infrastructure (docker) with the knowledge gained throughout this book.

8

NGINX within a Cloud Infrastructure

In the evolving landscape of web infrastructure, the shift from traditional server configurations to cloud architectures is undeniable. Traditional deployments, where servers were manually configured to host multiple sites or applications, are now giving way to more agile and scalable cloud-based methods. This chapter looks at the practical aspects of adopting a cloud infrastructure, highlighting the resource management efficiencies, enhanced security, and improved management of high traffic demands it offers.

In this chapter, we will focus on the role of NGINX within a cloud infrastructure, using Docker as the platform of choice. By integrating NGINX with Docker, we'll demonstrate how to orchestrate multiple services seamlessly, fostering a resilient and adaptable environment. Whether Docker is a new concept to you or not, rest assured that we'll guide you through your first Docker experience, with NGINX as a common thread, ensuring a smooth and understandable journey into containerized deployment.

This chapter covers the following topics:

- Understanding cloud infrastructure
- Using Docker
- Setting up NGINX inside Docker
- Setting up NGINX inside Docker to proxy host applications

Understanding cloud infrastructure

Cloud services are an integral part of our online existence. Our mobile applications synchronize seamlessly with their desktop counterparts, and the server once familiar to us has transformed into today's cloud—a concept that simplifies life for users while making web infrastructure more complex. This digital metamorphosis has revolutionized the way we store and interact with our data, moving us toward an always-connected, cloud-centric reality.

The traditional approach

To comprehend the intricacies of cloud infrastructure, let's revisit traditional setups with a modern *cloud* lens. Consider the case of WordPress, the renowned open source platform for website creation, which runs on a web server, a PHP server, and a database server. The classic method is to install each component on a single machine—a simple solution for hosting a solitary blog:

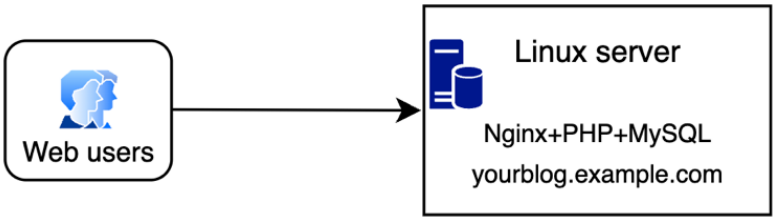


Figure 8.1: A traditional approach to running your web project

However, scaling this model reveals its limitations. Various software dependencies may require multiple, sometimes contradictory versions, making coexistence difficult. What's more, as services expand, managing them on a single server becomes more and more complicated.

The cloud approach

In the pursuit of hosting multiple services on a single server, we often encounter a common problem: **conflicting software dependencies**. Imagine needing to run two applications, one requiring PHP 7.4 and the other PHP 8.3. Docker addresses this by encapsulating applications in containers, isolated environments that operate independently of the underlying Linux distribution's software versions.

Advantages of using Docker containers include the following:

- **Version management**: Docker containers simplify the process of maintaining and changing software versions, making it easy to go back if necessary
- **Isolation and efficiency**: Containers isolate applications, which not only improves portability for easier migration but also ensures that the resource consumption of one doesn't affect the others
- **Deployment streamlining**: By overcoming the Docker learning curve, deployment and scaling become easier to manage, offering a degree of flexibility not found in traditional configurations

Let's revisit the blog hosting scenario, now using Docker:

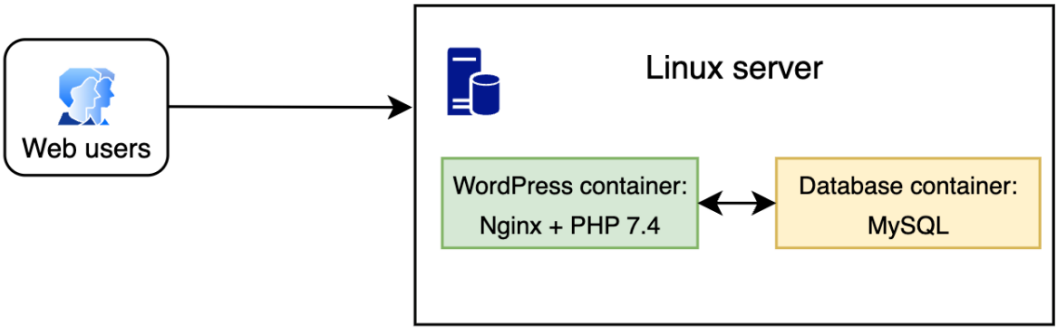


Figure 8.2: An example of how it would look running WordPress with Docker

The server runs a Linux distribution and, on top of this, two separate containers act as minimal Linux environments, each carrying only what's needed to run its respective applications.

The addition of a service such as Nextcloud, which could run PHP 8.3, alongside WordPress on PHP 7.4, demonstrates the ability of Docker:

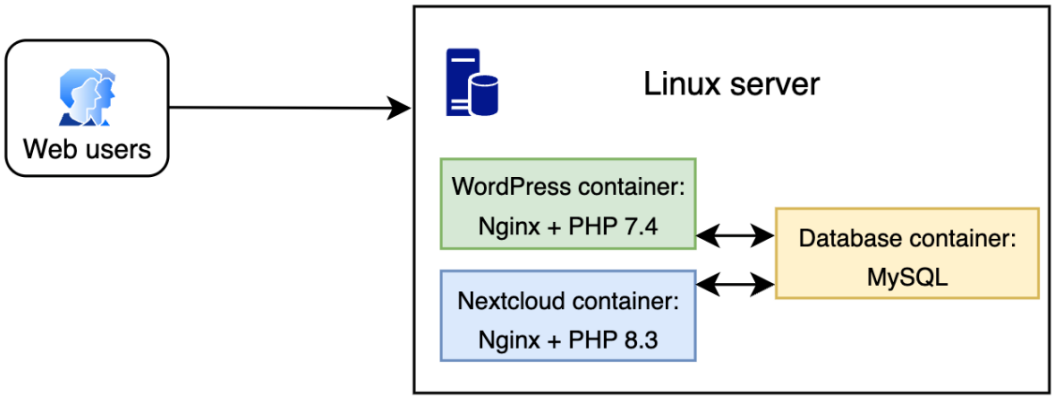


Figure 8.3: An example of how it would look running multiple containers with Docker

As shown, this configuration allows many Docker images, each with its own specific software version, to run simultaneously in an isolated environment. If a security problem occurs in one container, its effects are confined, protecting the rest of the system.

This chapter won't cover advanced topics such as Docker Swarm for **high availability** (**HA**) or Kubernetes for scalability, but these are important areas to explore for the advancement of cloud infrastructure.

Using Docker

In the previous section, we explored the fundamentals of cloud architectures. We're now going to move on from theory to practice by installing and configuring Docker step by step to launch our first container.

Docker is not just a tool but a paradigm shift—a new era where software can be packaged and isolated, ensuring consistency across environments. By the end of this section, Docker will be more than a concept; it will be an integral part of your toolbox, starting with the deployment of an NGINX container.

Installing Docker

Fortunately for us, the Docker team has provided a script that simplifies the installation process. This script is compatible with a range of Linux distributions, including **Red Hat Enterprise Linux** (**RHEL**), CentOS, Fedora, Debian, Ubuntu, and their derivatives.

To install Docker, you will need to run the script with root privileges. Open your terminal and enter the following command:

```
# curl -s https://get.docker.com | bash
```

With this single command, Docker will be installed natively on your system using custom repositories adapted to your Linux distribution's package manager. This means that updating Docker will be as easy as updating any other package on your system.

Your first Docker container

There are many ways to operate Docker containers. For instance, you can launch a container using a simple command:

```
root@docker:~# docker run -d nginx
latest: Pulling from library/nginx
e1caac4eb9d2: Pull complete
```

This command pulls the NGINX image from Docker Hub and starts a new container in detached mode. However, this container runs with default settings and isn't yet configured for specific use.

To customize the container, you can pass additional parameters. For example, to map the container's port 80 to the host's port 80, allowing web traffic to reach the container, you would run the following:

```
root@docker:~# docker run -d nginx -p 80:80
```

The first command launches the NGINX container with the default configuration. The second command runs the same container, but now it is accessible via the host machine's port 80. But what if you require a more advanced and complete configuration? This is where Docker Compose steps in, offering a solution for managing multi-container Docker applications with ease.

In the context of this book, the NGINX image serves as an excellent example of Docker's capabilities as it showcases how containerization simplifies the deployment of services that traditionally require dedicated servers.

As we explore deeper, we'll see how to tailor an NGINX container to serve static content or act as a reverse proxy, introducing the concept of Docker volumes and how to use them to serve custom configuration files and content.

Simplifying with Docker Compose

After running our first Docker container, it became clear that managing a container's parameters directly from the command line can quickly become cumbersome. **Docker Compose** simplifies this process by allowing us to define and run multi-container Docker applications using a YAML file for configuration.

Let's create an equivalent setup to the one we ran in the previous section—an NGINX container with port 80 exposed to the host machine:

1. Begin by creating a `/root/nginx` directory, and within that directory, save a file named `docker-compose.yml` with the following content:

```
version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
```

> **A reminder**
> Although the file format is called *YAML*, the extension needs to be `.yml`.

   In this `docker-compose.yml` file, we've defined a service named `nginx`, using the official `nginx` image tagged with `latest`, which means it will be updated every time we pull it from Docker Hub. If you wish, you can specify a fixed version, such as `nginx:1.25.4`. More details can be found on the Docker Hub for NGINX page (`https://hub.docker.com/_/nginx/`).

2. Now, with the Docker Compose file saved, run the following command in the same directory:

```
root@nginx:~/nginx# docker compose up
[+] Running 1/1
 ✓ Container nginx-nginx-1  Created
```

3. You've just launched your first container using Docker Compose. To stop this container, simply use the *Ctrl* + *C* shortcut (`^C`). Alternatively, you can start the container in detached mode with `up -d` and stop it with `down`:

```
root@nginx:~/nginx# docker compose up -d
root@nginx:~/nginx# docker compose down
```

With Docker Compose, running NGINX in Docker becomes a matter of defining the desired state in a file, which is easier to manage and read than standalone commands. In the next section, we'll explore how to further configure NGINX in Docker, tailoring it to our specific needs.

Setting up NGINX inside Docker

Having become familiar with Docker Compose, we're set to advance our container usage. In this section, we'll improve our NGINX container by adding personalized configurations and website content.

Let's begin by revisiting the `docker-compose.yml` file we created earlier. This file already specifies the NGINX service and maps the container's port 80 to port 80 on the host machine, making the web server accessible from the host.

Next, we want our NGINX container to serve our website using the actual configuration files and content from our host machine. To achieve this, we'll use Docker volumes. Volumes are the preferred mechanism for persisting data generated and used by Docker containers. Here's how you can modify the `docker-compose.yml` file to mount a volume:

```
version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - «80:80»
    volumes:
      - ./config/nginx.conf:/etc/nginx/nginx.conf
      - ./html:/usr/share/nginx/html
```

In this setup, `./config/nginx.conf` is the path to your custom NGINX configuration file on your host machine. `/etc/nginx/nginx.conf` is the path where the NGINX container expects to find the configuration file. The second volume, `./html`, is the directory on your host machine that contains your website's content, mounted into `/usr/share/nginx/html/` in the Docker container.

This approach ensures that you can edit your NGINX configuration or website content directly on your host machine, and those changes will be reflected inside the container.

A note

NGINX is preconfigured; therefore, you can skip mounting the `nginx.conf` volume and use the default settings. Do mount the `html` directory as it is where NGINX expects to find web content to serve.

Integrating PHP with NGINX using Docker Compose

As we progress, our next step is to elevate our NGINX server to handle dynamic content with PHP. This combination is common in the web development world, and Docker Compose makes it easy to implement.

We'll start by extending our existing `docker-compose.yml` file to include a PHP service, and then we will ensure NGINX can communicate with the PHP service using FastCGI:

1. First, we need to add a PHP service that runs **FastCGI Process Manager** (`php-fpm`). Here's how the updated `docker-compose.yml` file might look:

```
version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./config/nginx.conf:/etc/nginx/nginx.conf
      - ./html:/usr/share/nginx/html
    depends_on:
      - php
  php:
    image: php:8.3-fpm
    volumes:
      - ./html:/usr/share/nginx/html
```

   In this setup, the `php` service uses the official PHP image with `php-fpm`. We also mount the `html` directory into the PHP container to ensure it has access to the same web files as NGINX.

2. Before starting the Docker containers, we need to make sure NGINX and PHP communicate. Let's edit the `nginx.conf` file to include FastCGI parameters. Here's an example of `nginx.conf` with included FastCGI parameters:

```
events {}
http {
  server {
    listen 80;
```

```
    server_name localhost;

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

    location / {
        try_files $uri $uri/ =404;
    }

    location ~ \.php$ {
      try_files $uri =404;
      fastcgi_pass php:9000;
      fastcgi_index index.php;
      fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_
script_name;
      include fastcgi_params;
    }
  }
}
```

This configuration instructs NGINX to forward requests for PHP files to the PHP-FPM service running in the `php` container.

> **Note**
>
> NGINX resolves the `php` service name using Docker's internal DNS to communicate with the `php-fpm` service. If you rename the service in your `docker-compose.yml` file, remember to update the `nginx.conf` file accordingly to match the new service name.

3. With this `docker-compose.yml` and `nginx.conf` configuration, you can run `docker compose up`, and NGINX will serve both your static content and your dynamic PHP pages:

```
root@nginx:~/nginx# docker compose up
[+] Running 2/0
 ✓ Container docker-php-1    Created
 ✓ Container docker-nginx-1  Created
Attaching to nginx-1, php-1
php-1 [02-Mar-2024 11:42:30] NOTICE: fpm is running, pid 1
```

4. After launching your containers, to check that NGINX and PHP are interacting correctly, create a PHP file containing `<?php phpinfo(); ?>`. This will display a PHP information page when accessed, confirming successful communication between the containers:

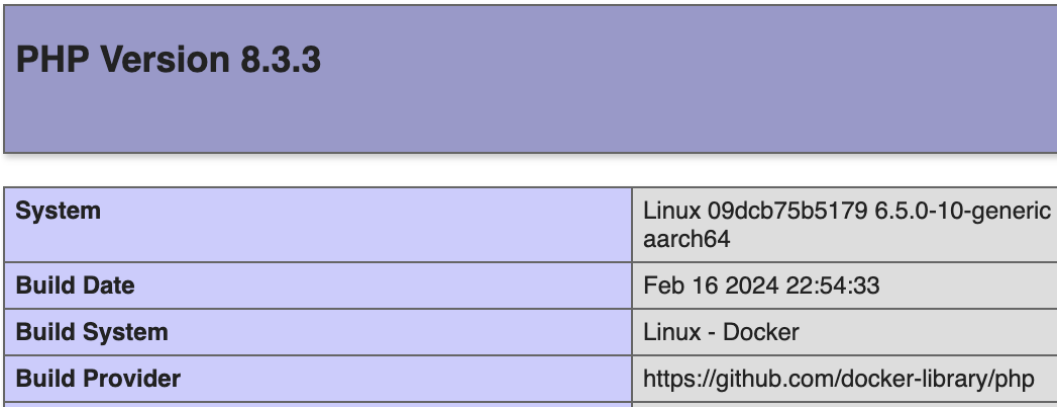


Figure 8.4: phpinfo running within Docker

This subsection has guided you through adding a PHP service to your Docker environment and configuring NGINX to process PHP scripts via FastCGI. This configuration mimics a production environment, providing a solid foundation for building and deploying PHP applications with Docker.

In the next section, we'll explore how to configure a Docker-contained NGINX to proxy applications running directly on the host server.

Setting up NGINX inside Docker to proxy host applications

Fortunately for us, Docker is built with inherent support for container-to-host connections. When setting up NGINX inside a container, a special `host.docker.internal` hostname is used to target the host machine.

Here's a sample NGINX proxy configuration for interfacing with a web application hosted on the server:

```
server {
    listen 80;

    location /hostapp {
        proxy_pass http://host.docker.internal:4000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

To ensure NGINX can communicate with the host server's web application, check that the host's firewall allows incoming traffic on the specified ports.

Concluding this section, we've successfully bridged the containerized NGINX with host-resident applications, leveraging Docker's internal networking features.

In the next section, we'll recap what we learned in this book.

Summary

In this chapter, we've taken our first steps with Docker and Docker Compose, set up an NGINX container, added PHP, and learned to proxy traffic to a web app on the host server.

Combined with the knowledge from the previous chapters, the exercises in this chapter have highlighted NGINX's potential as a key asset in cloud infrastructure, strengthening security and managing traffic efficiently, well suited for SSL handling, load balancing, and content caching.

In the next chapter, we'll learn how to deploy and update NGINX with automated tasks using orchestration tools.

9

Fully Deploy, Manage, and Auto-Update NGINX with Ansible

In the previous chapter, we discussed how NGINX works within a cloud infrastructure and how to use Docker Compose configuration files. Beyond the cloud, configuration management stands out for its ability to strengthen the resilience of any infrastructure and streamline compliance with security standards.

In this chapter, we'll dive into the practical uses of Ansible, giving you a hands-on tour of its capabilities. You'll learn to employ Ansible for efficient configuration management, allowing you to deploy NGINX with precision on any number of servers, from a single machine to an entire fleet, all with a unified configuration approach.

This chapter covers the following topics:

- Understanding configuration management
- Your first steps with configuration management using Ansible
- Setting up NGINX using Ansible
- Setting up automatic updates using Ansible

Understanding configuration management

Configuration management is like having a master key for your infrastructure's setup, especially when dealing with multiple deployments. We've seen a glimpse of this with Docker Compose, where all configurations live in Docker, ready to be reused. This approach isn't just tidy; it ensures every part of your system walks in step, making it easy to scale or clone configurations when needed. It's all about saving time, staying organized, and automating wherever possible.

Stepping into configuration management, we turn to **Ansible**, a tool celebrated for its simplicity and efficiency. Unlike other management tools that require installing agents on each server, Ansible operates over SSH, minimizing setup overhead and maintaining a lightweight presence. It is an agentless orchestration tool that brings simplicity and security to the forefront of server automation. While our tutorials will primarily focus on Ubuntu due to its widespread adoption, the principles and practices are transferable to other distributions such as Debian, RHEL, or Fedora, with minor adjustments to adapt to different package managers and system layouts.

We now have a better idea of what configuration management is. In the next section, we will run our first playbook with Ansible.

Running your first Ansible playbook

Ansible is an orchestration tool organized as playbooks. Each playbook can have multiple roles. We'll start with installing Ansible, to run our first playbook for installing NGINX.

Ansible orchestrates tasks through playbooks composed of multiple roles. Our initial step is installing Ansible to execute our first role: *installing Nginx*.

Let's begin with the installation of Ansible. Ansible can be set up on either a remote server or your local Linux system, as long as there is connectivity to your NGINX server:

1. We'll be using the root account every step of the way:

```
root@ansible:~# apt install ansible
```

2. Ensure Ansible is installed by checking its version:

```
root@ansible:~# ansible --version
ansible 2.10.8
```

3. Now it's time to create your first Ansible role to install NGINX. Ansible roles have a standard directory structure. Use the `ansible-galaxy` command to create it:

```
root@ansible:~# ansible-galaxy init nginx_install
- Role nginx_install was created successfully
```

   This will create a directory called `nginx_install` with subdirectories for tasks, handlers, templates, and more.

4. Next, create the path `roles/nginx_install/tasks` with the `mkdir -p roles/nginx_install/tasks` command, then create the `roles/nginx_install/tasks/main.yml` file with the following content:

```
---
- name: Install nginx
  apt:
    name: nginx
```

```
    state: latest
    update_cache: yes
```

This will define a single task to install NGINX using Ubuntu's package manager.

5. Next, we'll create a playbook to use the role. At the root of your `nginx_install` project, create the `nginx_install.yml` file, with the following content:

```
---
- hosts: webservers
  become: true
  roles:
    - nginx_install
```

This will apply the role to all the hosts set (group: `webservers`).

6. Our last file needed for the playbook is the inventory. This is where your machine's addresses are stored. Create the `inventory.ini` file with the following content:

```
[webservers]
server_ip ansible_ssh_user=root
```

Please make sure to use an SSH key and replace `server_ip` with the IP of your server in order to connect to the server.

7. In this inventory, we created a group of hosts called `webservers`. Within this group, we added one server, `server_ip`. For scenarios involving multiple servers, your inventory would expand to list each one under this group, allowing for parallel management across your infrastructure. Here's another example of `inventory.ini` with multiple servers:

```
[webservers]
server1.example.com ansible_ssh_user=root
server2.example.com ansible_ssh_user=root
[backend_api]
api1.example.com ansible_ssh_user=root
api2.example.com ansible_ssh_user=root
```

8. We now have all the files needed to run our first playbook. Let's run the playbook:

```
root@ansible:~/nginx_install# ansible-playbook -i inventory.ini
nginx_install.yml
PLAY [all]
TASK [Gathering Facts]
ok: [testvm.lxd]
TASK [nginx_install : Install nginx]
changed: [testvm.lxd]
PLAY RECAP
testvm.lxd: ok=2    changed=1
```

Great, our first Ansible playbook has successfully installed NGINX. If you run the playbook again, it will recognize that NGINX is already installed and won't attempt a reinstallation.

You can now access your web server on port `80` to check whether NGINX is available:

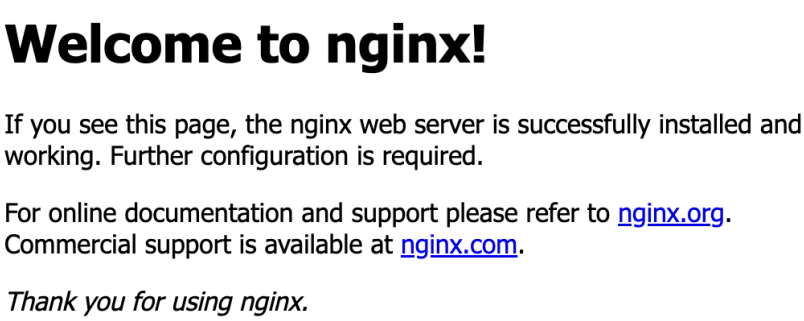


Figure 9.1: Default Nginx configuration installed by Ansible

We've installed Ansible, and installed NGINX on a remote server using Ansible. In the next section, we'll learn how to handle configuration files using Ansible.

Setting up NGINX using Ansible

In the last section, we made a project called `nginx_install`. We'll continue with this project in order to have Ansible write the `nginx.conf` file for us. So let's get started:

1. First, create a directory at `roles/nginx_install/files` with the `mkdir -p roles/nginx_install/files` command, then add your `nginx.conf` file to this folder.
2. Then, make a new task to copy the `nginx.conf` file. Edit the existing `roles/nginx_install/tasks/main.yml` file and add the following:

```
- name: Copy nginx configuration file
  copy:
    src: nginx.conf
    dest: /etc/nginx/nginx.conf
    owner: root
    group: root
    mode: '0644'
  notify: restart nginx
```

This will send the `nginx.conf` file from your `files` directory to the `/etc/nginx/` directory on your remote server.

3. Next, we'll define a handler within Ansible. Handlers are tasks that only run when notified by another task. In our case, we'll set up a handler to restart NGINX, but it will only trigger if there's been a change to the configuration file.

   Let's create our handler by creating the `roles/nginx_install/handlers` directory with the `mkdir -p roles/nginx_install/handlers` command. Then, let's create a `roles/nginx_install/handlers/main.yml` file with the following content:

   ```
   - name: restart nginx
     service:
       name: nginx
       state: restarted
   ```

4. Let's run the playbook again:

   ```
   root@ansible:~/nginx_install# ansible-playbook -i inventory.ini
   nginx_install.yml
   PLAY [all]
   TASK [Gathering Facts]
   ok: [testvm.lxd]
   TASK [nginx_install : Install nginx]
   ok: [testvm.lxd]
   TASK [nginx_install : Copy nginx configuration file]
   changed: [testvm.lxd]
   RUNNING HANDLER [nginx_install : restart nginx]
   changed: [testvm.lxd]
   PLAY RECAP
   testvm.lxd: ok=4    changed=2
   ```

   The output shows the `Install nginx` task as `OK` since NGINX is already in place from a previous run. The `copy nginx configuration file` task was successful, prompting the handler to restart NGINX. If you rerun the playbook, it won't repeat these actions as it recognizes NGINX's installation and the existing configuration.

NGINX has been installed with Ansible. In the next section, we will learn about automatic updates and how to secure your server environment directly with Ansible configuration files.

Setting up automatic updates using Ansible

unattended-upgrades is a valuable tool that ensures your system automatically receives updates in response to security vulnerabilities, such as CVE exploits. It's particularly useful for critical applications such as NGINX, where staying ahead of potential threats is crucial. When a security update is issued for NGINX, `unattended-upgrades` takes care of the update process, providing peace of mind that your server remains secure without manual intervention.

To install the `unattended-upgrades` package, you can edit the existing file `roles/nginx_install/tasks/main.yml` and add the following:

```
- name: Install unattended-upgrades
  apt:
    name: unattended-upgrades
    state: present
```

Rerun the playbook to install the package:

```
root@ansible:~/nginx_install# ansible-playbook -i inventory.ini nginx_
install.yml
[…]
TASK [nginx_install : Install unattended-upgrades]
ok: [testvm.lxd]
```

With `unattended-upgrades` installed, your system is configured for automatic security updates. For those seeking comprehensive updates beyond security, the package's documentation provides guidance for fine-tuning your settings.

Summary

In this chapter, we explored the power of configuration management with Ansible, learning to orchestrate our NGINX setup seamlessly. Through Ansible's capabilities, we've initiated package installations, updated configurations, and embraced the efficiency of automated updates. The scalability of these processes has opened new doors for system administration. While we've only scratched the surface, further exploration into Ansible can unlock automated deployments of dynamic configurations, TLS certificates, comprehensive firewall setups, and much more. For those interested in deepening their understanding of Ansible, a variety of playbooks can be found at `docs.ansible.com`.

In the next chapter, we'll explore the practical applications and real-world scenarios for implementing NGINX.

10
Case Studies

Throughout this book, we've navigated the broad capabilities of NGINX, from delivering static content to implementing intricate load-balancing strategies. Your newfound knowledge extends from basic NGINX setup on servers to tailor-fit configurations for diverse web applications, enhanced by a variety of advanced modules that unlock complex features.

Now, it's time to apply what we've learned through a series of real-world scenarios. Securing communications with HTTPS will be our foundational step, ensuring all interactions with your web services are encrypted by default. With this security measure in place, we'll progress to setting up a complete WordPress site. Here, we'll not only focus on getting your site live with NGINX but also on utilizing the secure channels we've established, integrating best practices for optimization and caching to enhance performance.

As we progress, the chapter will guide you in deploying a Nextcloud instance, illustrating how to establish a secure, private cloud storage solution.

This chapter covers:

- SSL certificates and HTTPS by default
- Implementing HTTP/2
- Deploying WordPress, efficient content management
- Deploying NextCloud, a robust personal cloud service

Exploring SSL Certificates and HTTPS by default

Digital security is paramount, adopting a *security-first* approach is not just best practice—it's essential. As such, plaintext transmission is no longer acceptable; SSL encryption is the new standard.

Recognizing this, we turn to **acme.sh**, a versatile tool favored for its flexibility in supporting a diverse range of certificate authorities and its capability to employ DNS APIs for the creation of wildcard certificates. To streamline our setup, we'll craft a centralized NGINX configuration file for SSL, `ssl.conf`, which will serve as a reusable component in all our secure site configurations. This foundation of SSL by default sets a robust stage for all web services we deploy.

Certificate Management with acme.sh

Step by step, we'll learn how to exploit acme.sh' integration with DNS APIs, simplifying the validation process for certificate issuance. By the end of this tutorial, you'll have the knowledge you need to obtain your first signed wildcard certificate.

Let's begin with the installation of acme.sh:

```
root@nginx:~# curl https://get.acme.sh | sh
```

Then add the alias "`alias acme.sh=~/.acme.sh/acme.sh`" into the file "/root/.bashrc"

Verify acme.sh has been installed properly. First, make sure to close and reopen your terminal (or ssh session), as advised by the installation script:

```
UTC 2024] Installing to /root/.acme.sh
UTC 2024] Installed to /root/.acme.sh/acme.sh
UTC 2024] Installing alias to '/root/.profile'
UTC 2024] OK, Close and reopen your terminal to start using acme.sh
UTC 2024] Installing cron job
```

Figure 10.1: acme.sh script inviting us to close and reopen the terminal session.

Then run `acme.sh --version`:

```
root@nginx:~# acme.sh --version
https://github.com/acmesh-official/acme.sh
v3.0.8
```

Acme.sh has now been installed. We'll be using acme.sh to generate a wildcard certificate but before we do so, let's have a quick look at what is a wildcard certificate, why we want one, and how acme.sh will help us generate one.

Imagine the following: we own the domain `example.com`. Using acme.sh, we'll be generating one single certificate which will be valid for multiple domains. The first domain will be `example.com` and the second domain will be `*.example.com`. There are many reasons why we want to do this:

- First, to make things easier. No need to adapt the NGINX configuration to run on port 80 with specific folders. Once you have the certificate for your domain, you know you can use it for a subdomain, and reuse the same certificate for a different subdomain.
- Second, to enhance the security of your infrastructure by preventing scanners. Each signed certificate is publicly available and you can find them with tools like `https://crt.sh`.

To generate wildcard certificates, the fastest way is through DNS API. This means that you will be creating a temporary subdomain to let the certificate authority know you own the domain. For example, you will be creating "yes-i-own-it-12345.example.com" and in exchange, the certificate authority will issue a certificate. We'll learn how to do that automatically, by giving acme.sh the right to create temporary subdomains. Once set, your certificates will be renewed automatically, every 90 days.

In this example, we will be using the Cloudflare API but know that acme.sh supports many more registrars, over a hundred registrars. The list is available in their official github repo, here: `https://github.com/acmesh-official/acme.sh/wiki/dnsapi`.

acme.sh and the DNS API

Using Cloudflare, let's get the API key:

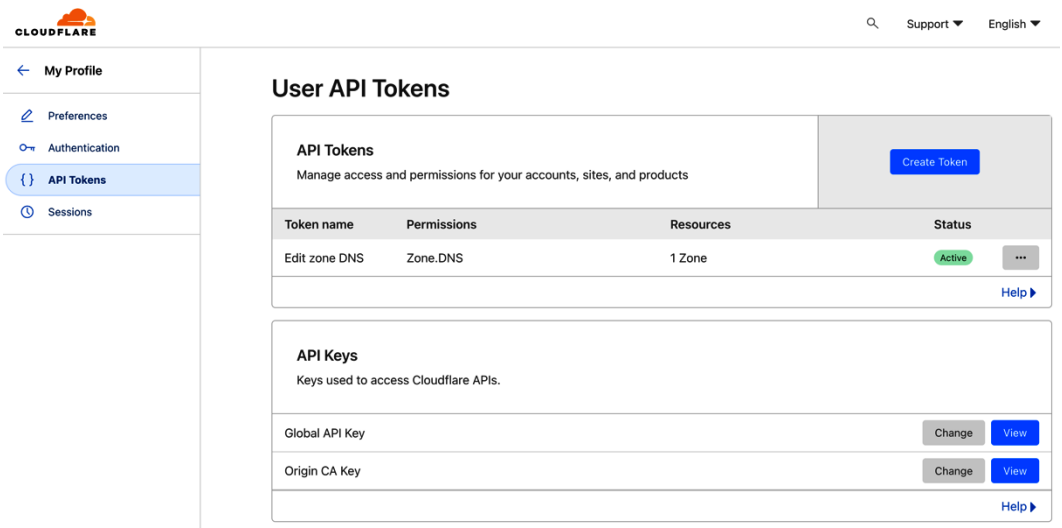


Figure 10.2: Obtaining the Cloudflare API key

As shown above, you can get the API key by going to **Profile** and then selecting **View** for **Global API Key**

For this example, let's say our API Key is `abcd1234`. We will store it in `/root/.acme.sh/account.conf` alongside the email address used for your Cloudflare account:

```
SAVED_CF_Key='abcd1234'
SAVED_CF_Email=my-cloudflare-account@personal.email
```

Issuing a signed certificate

We're now set to have our first signed certificate. We've installed acme.sh and we've given acme.sh the right to access our domain `example.com`. Our next step is to use acme.sh to issue a certificate:

```
root@nginx:~# acme.sh --issue --dns dns_cf -d example.com -d
*.example.com
```

This command will make acme.sh contact Cloudflare to create temporary subdomains, then have a Certificate Authority (default: zerossl) sign the certificate it just generated, and will delete the temporary subdomains.

```
UTC 2024] Your cert is in  /root/.acme.sh/example.com/example.com.cer
UTC 2024] Your cert key is in  /root/.acme.sh/example.com/example.com.key
UTC 2024] The intermediate CA cert is in  /root/.acme.sh/example.com/ca.cer
UTC 2024] And the full chain certs is there:  /root/.acme.sh/example.com/fullchain.cer
```

Figure 10.3: Confirmation that acme.sh generated and issued a signed certificate.

Congratulations on your first signed wildcard certificate! In the next subsection, we will be centralizing NGINX' SSL configuration within one configuration, using our existing certificate.

Centralizing SSL Configuration with NGINX

In the last subsection, we've learned how to obtain signed certificates. We're going to use these certificates in the NGINX configuration. The challenge we have is that depending on our setup, we might need to use the certificate in multiple places as it is valid for multiple subdomains. In order to avoid repetition, and to manage our configuration in the easiest way possible, we'll write a configuration file that we will include later into other configuration files.

To begin, let's write our file `ssl.conf`. This file has to be stored within the NGINX configuration folder, such as `/etc/nginx/` for example:

```
  ssl_certificate /root/.acme.sh/example.com/fullchain.cer;
  ssl_certificate_key /root/.acme.sh/example.com/example.com.key;
  ssl_protocols TLSv1.3;
  ssl_prefer_server_ciphers off;
```

```
ssl_session_cache  builtin:1000  shared:SSL:10m;
ssl_session_tickets on;
ssl_stapling on;
ssl_stapling_verify on;
```

And that's it! As we will deploy new projects, we will include this `ssl.conf` file to save time.

In the next section, we will introduce HTTP/2.

Implementing HTTP/2 with SSL

HTTP/2 stands as a significant evolution of the HTTP protocol, designed to enhance web performance and efficiency. With features like multiplexing, which allows for multiple requests and responses over a single connection, and header compression to minimize overhead, HTTP/2 makes web browsing noticeably faster and more resource-efficient. Another key feature is server push, where servers preemptively send resources to the client without waiting for a request, further optimizing load times.

Here's how you can enable HTTP/2 in your server block:

```
server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;
…
}
```

Be sure to activate HTTP/2 when setting up SSL, it is well-supported by modern browsers but requires a secure connection.

Although HTTP/3 offers promising advancements, it's not quite ready for widespread production use just yet. We encourage you to keep an eye on its development as we believe it will be exciting to implement when it's fully ready.

Coming next, we'll walk through setting up a **WordPress site** with secure certificates and HTTP/2 enabled.

Deploying a WordPress site

WordPress is currently the most popular content management system on the entire web. According to Kinsta (`https://kinsta.com/wordpress-market-share/`), its market share totals 42%. For a lot of web server administrators, setting up WordPress sites or blogs has become a common task, whether it is for personal or professional use.

Preparing your server and obtaining WordPress

In this section, we will be getting your server ready for downloading and installing the WordPress application. There will be a few configuration files to go through to make sure WordPress runs smoothly.

System requirements

The first step you need to go through to set up a WordPress site on a fresh new server is to make sure you have the necessary components installed and up to date: it is recommended that you run at least PHP 8.1 and MySQL Server 8. If you haven't done so yet, running the following commands will provide a basic working environment with minimal PHP extensions. Under a Debian-based Linux operating system:

```
# apt install mysql-server php8.1-fpm php8.1-mysql php8.1-gd php8.1-
xml php8.1-mbstring php8.1-curl php8.1-zip
```

If your server runs a Red Hat-based OS, such as Fedora:

```
# dnf install mysql-server php8.1-fpm php8.1-mysqlnd php8.1-gd php8.1-
xml php8.1-mbstring php8.1-curl php8.1-zip
```

If you have an older version installed on your system, it is recommended that you upgrade to the latest available version using the `apt update && apt upgrade` or `dnf upgrade --refresh` commands.

PHP configuration

After making sure your server components meet the minimum requirements, you should edit some of the settings, if you want WordPress to run smoothly. There are two main aspects of the PHP configuration you should look into. First, the default PHP configuration file (`php.ini`) contains directives that you will probably want to update:

- `cgi.fix_pathinfo`: Set this value to `0` for security reasons, as we have seen in *Chapter 5*.
- `post_max_size`: By default, the maximum size of the `POST` request body is `8` megabytes. Increase the value if necessary; keep in mind that file uploads are usually performed via `POST` requests.
- `upload_max_filesize`: Set to `2` megabytes by default, this will need to be increased if you want to allow uploading of large files.
- `date.timezone`: You will get a warning if you leave this blank as it is by default. Refer to `https://php.net/manual/en/timezones.php` to find out the proper value in your situation.

The second aspect of the configuration is the PHP-FPM side. The main `php-fpm.conf` file does not require immediate changes, however, if you haven't done so yet, you will need to create a *configuration pool*: a set of configuration directives that apply to a particular website or application. This allows you to run the PHP processes under a specific user account, and optionally configure a specific network interface for communicating with NGINX.

Create a new pool by declaring its name between brackets:

```
[wordpress]
```

Append the following configuration directives:

```
; Specify user account and group for the pool
; We assume that you created a "wordpress" user and group
user=wordpress
group=wordpress
; Network interface and listening port
; Use 127.0.0.1 if Nginx runs on the same machine
listen=127.0.0.1:9000
; Only allow connections from local computer
; Change this value if Nginx runs on a different machine
allowed_clients=127.0.0.1
```

Optionally, you may enable *chrooting*: specify a root directory for the PHP processes of this pool. For example, if you set the chroot to `/home/wordpress/www`, your PHP scripts will only be able to read files and directories within the specified path (any attempt to read or write a file or directory outside of `/home/wordpress/www` will systematically fail). It is highly recommended you enable this feature: should a security breach be discovered in the WordPress code, attackers would only be able to exploit files within the reach of your PHP process; the rest of your server would not be compromised:

```
chroot /home/wordpress/www;
```

Other configuration directives are documented at length in the default pool file supplied with PHP-FPM; their default values are suitable in most cases.

MySQL configuration

At the time of installing MySQL server, you were asked to set up administrator (`root`) credentials. Since these credentials allow full access to the SQL server, including permissions on all databases, you should never use them in any of your PHP applications. The best practice is to create a separate MySQL user and to assign permissions on the database that will be used by your application:

1. Log in to your local MySQL server with the following command:

```
# mysql -u root -p
```

2. Create a new SQL database:

```
mysql> CREATE DATABASE wordpress;
```

3. Create a SQL user and grant all permissions to the `wordpress` database (don't forget to specify a complex enough password):

```
mysql> GRANT ALL PRIVILEGES ON wordpress.* TO
'wordpress'@'localhost' IDENTIFIED BY 'password';
```

4. Now, run the `exit` command to leave the MySQL console and try logging in to the server using the newly created account:

```
mysql> exit
# mysql -u wordpress -p
mysql> SHOW DATABASES;
```

   You should see the `wordpress` database you created a minute ago.

Downloading and extracting WordPress

The last step is to download the latest version of WordPress and extract it at the location specified earlier; in our example: `/home/wordpress/www`. The latest version can always be found at `https://wordpress.org/latest.tar.gz`:

```
/home/wordpress/www# wget https://wordpress.org/latest.tar.gz
/home/wordpress/www# tar xzf latest.tar.gz
/home/wordpress/www# mv ./wordpress/* ./ && rm -r ./wordpress
```

Make sure the user and group are properly set, and give write permissions to the `wordpress` user over the application files:

```
/home/wordpress/www# chown -R wordpress ./
/home/wordpress/www# chgrp -R wordpress ./
/home/wordpress/www# chmod -R 0644 ./
```

NGINX configuration

Before you can begin setting up WordPress via the user-friendly web installer, you will need to finalize your NGINX server configuration. We will go down to every last detail in the following subsections.

HTTP block

We will be going down the blocks starting at the top level: the HTTP blocks, encompassing directives that have an effect on the entire server. This implies that the directives placed here will affect all of the websites served by this instance of NGINX. Open your NGINX main configuration file (`nginx.conf`) and insert or update the following directives:

```
# Sets the user and group under which the worker processes
# will run. The following values are valid assuming your server
# will only be hosting one website.
user wordpress wordpress;
```

```
worker_processes 8; # 1 process per core
pid /var/run/nginx.pid;

events {
# Edit this value depending on your server hardware
   worker_connections 768;
}

http {
   # Core settings affecting I/O
   sendfile on;
   tcp_nopush on;
   tcp_nodelay on;

   # Default Nginx values
   keepalive_timeout 65;
   types_hash_max_size 2048;
   include /etc/nginx/mime.types;
   default_type application/octet-stream;

   # Set access and error log paths
   access_log /var/log/nginx/access.log;
   error_log /var/log/nginx/error.log;

   # Enable gzipping of files matching the given mime types
   gzip on;

   gzip_types text/plain text/css application/json application/x-
javascript text/xml application/xml application/xml+rss text/
javascript;

   # Include virtual host configuration files;
   # Edit path accordingly
   include /etc/nginx/sites-enabled/*;
}
```

Server block

The following step will require you to create a new file in the directory specified earlier. For example, create a file called `wordpress.conf` in the `/etc/nginx/sites-available/` folder. Define your virtual host configuration by inserting or updating the following directives:

```
server {
#Default server on port 80
  Listen [::]:80 default_server;
```

```
  listen 80 default_server;
  server_name _;
#Always redirect to https
  return 301 https://$host$request_uri;
}

server {
   # Listen on all network interfaces on port 443 SSL and HTTP2
   listen 443 ssl http2;
   listen [::]:443 ssl http2;
   #Include ssl.conf created earlier
   include ssl.conf

   # Specify the host name(s) that will match the site
   # The following value allows both www. and no subdomain
   server_name .example.com;

   # Set the path of your WordPress files
   root /home/wordpress/www;

   # Automatically load index.php
   index index.php;

   # Saves client request body into files, cleaning up afterwards
   client_body_in_file_only clean;
   client_body_buffer_size 32K;

   # Allow uploaded files up to 300 megabytes
   client_max_body_size 300M;

   # Automatically close connections if no data is
   # transmitted to the client for a period of 10 seconds
   send_timeout 10s;

   # The rest of the configuration (location blocks)
   # is found below
   [...]
}
```

Once done, create a symlink for /etc/NGINX/sites-available/wordpress.conf in /etc/NGINX/sites-enabled/ with the following command: `ln -s /etc/nginx/sites-available/wordpress.conf /etc/nginx/sites-enabled/`

Location blocks

Finally, set up your `location` blocks—directives that apply to specific locations on your site:

```
    # The following applies to static files:
    # images, CSS, javascript
    location ~* ^.+.(jpg|jpeg|png|gif|ico|css|js)$ {
          access_log off; # Disable logging
          # Allow client browsers to cache files
          # for a long period of time
          expires max;
    }

    # The following applies to every request
    location / {
       # Try serving the requested URI:
       # - If the file does not exist, append /
       # - If the directory does not exist,
       # redirect to /index.php forwarding the request URI
       # and other request arguments
          try_files $uri $uri/ /index.php?q=$uri&$args;
    }

    # The following applies to every PHP file
    location ~ .php$ {
          # Ensure file really exists
             if (!-e $request_filename) {
                   return 404;
             }
             # Pass the request to your PHP-FPM backend
             fastcgi_pass 127.0.0.1:9000;
          fastcgi_index index.php;
          fastcgi_param PATH_INFO $fastcgi_script_name;
          include fastcgi_params;
    }
}
```

WordPress configuration

Once your NGINX configuration is finalized and saved, make sure to reload the NGINX configuration, either via `systemctl reload nginx` or `/usr/local/nginx/sbin/nginx -s reload` (or your usual NGINX binary location).

If all goes well, you should be able to run the web-based WordPress installer by visiting `https://example.com/wp-admin/install.php` (replacing `example.com` by your own domain name). You will be prompted for:

- The name of the database you created earlier, in our example: `wordpress`
- The SQL username you created earlier, in our example: `wordpress`
- The password associated to the user: `password`
- The database host: `127.0.0.1`, if your MySQL server is hosted on the same server
- A prefix for all SQL tables created by WordPress: `wp_`

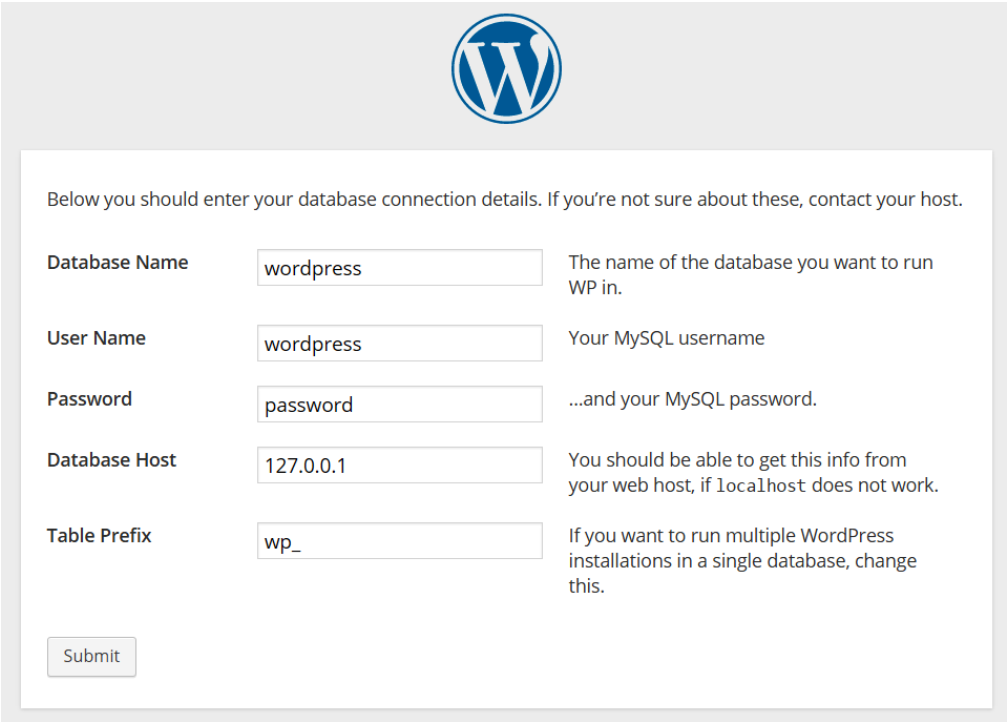


Figure 10.4: WordPress database installation page

Once the installer completes, you can begin configuring and preparing your WordPress site. In order to enable pretty URLs, you should check the **Settings** | **Permalinks** section: several URL schemes are offered, such as `https://example.com/post-name/` or `https://example.com/year/month/post-name/`.

…We've deployed Wordpress. In the next section, we'll be deploying Nextcloud alongside Wordpress using Docker to properly isolate our configurations for Wordpress and Nextcloud.

Deploying Nextcloud

Nextcloud represents the pinnacle of self-hosted cloud services, offering not just file storage, but also integrated calendars, contacts, and collaboration tools. As an open-source solution, it provides transparency and control, ensuring that your data is truly yours.

In this section, we will navigate the setup of Nextcloud on an NGINX server, emphasizing secure communication through validated SSL certificates. This will complement the multi-application hosting environment we've cultivated with NGINX, where it serves both the dynamic content management of WordPress and the robust data ecosystem of Nextcloud.

Getting Nextcloud

As we've already configured NGINX to serve our WordPress site, optimized specifically for WordPress's unique requirements, we'll now employ Docker to host Nextcloud. This approach ensures isolation and flexibility without disrupting our current setup. We'll use Docker's containerization to seamlessly integrate Nextcloud alongside WordPress. More exactly, we'll deploy the NextCloud All-In-One image which features an integrated Apache web server, and proxy it through NGINX. Please refer back to *Chapter 8* for guidance on setting up Docker if needed.

So, let's get started:

1. We'll create a directory `/root/nextcloud` and within this folder, we'll create the file `docker-compose.yml` with the official All-In-One Nextcloud Docker image. Please refer to their online github to get the latest docker compose file. Currently, it looks like this:

```
services:
  nextcloud-aio-mastercontainer:
    image: nextcloud/all-in-one:latest
    init: true
    restart: always
    container_name: nextcloud-aio-mastercontainer
    volumes:
      - nextcloud_aio_mastercontainer:/mnt/docker-aio-config
      - /var/run/docker.sock:/var/run/docker.sock:ro
    ports:
      - 8080:8080
      - "127.0.0.1:11000:11000"
    environment:
      - APACHE_PORT=11000
volumes:
  nextcloud_aio_mastercontainer:
    name: nextcloud_aio_mastercontainer
```

2. We've saved the file in our `directory /root/nextcloud`. Within this directory, let's start the container:

```
docker compose up -d
```

3. Now that our container is running, we'll need to add a server block into NGINX, in order to create a reverse proxy for Nextcloud. Fortunately for us, Nextcloud provides us with a configuration sample, available on their github repository (Nextcloud AIO reverse proxy). Here's how the NGINX configuration file should look like. We'll save this file into `/etc/nginx/sites-enabled/nextcloud.conf`:

```
server {
  listen 443 ssl http2;
  listen [::]:443 ssl http2;
  include ssl.conf;
  server_name nextcloud.example.com;
    location / {
        proxy_pass http://127.0.0.1:11000$request_uri;

        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_
for;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-Scheme $scheme;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Accept-Encoding "";
        proxy_set_header Host $host;

        client_body_buffer_size 512k;
        proxy_read_timeout 86400s;
        client_max_body_size 0;

        # Websocket
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
    }
}
```

4. Make sure to test your configuration and reload the NGINX server, before getting ready to set up Nextcloud:

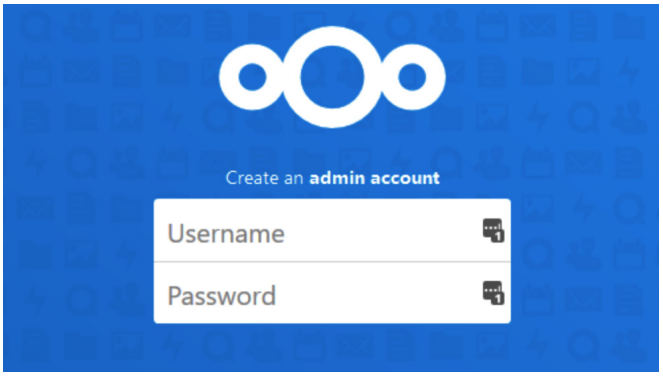


Figure 10.5: Nextcloud default installation page

There you have it, your Nextcloud server is ready for action. For further customization and detailed configurations, the Nextcloud official documentation is an excellent resource. You can find it at `nextcloud.com`.

With the foundations laid, you're now equipped to deploy numerous servers, utilizing Docker for containerization or installing directly on the host. Just remember to use compatible software versions.

In the next section, we'll recap the key points covered in this chapter.

Summary

Throughout this chapter, we've mastered the art of generating verified SSL certificates with acme.sh, tapping into DNS APIs for efficiency. We've explored the enhancements offered by HTTP/2 and laid the groundwork for HTTP/3, setting the stage for optimal web performance. This knowledge has empowered us to deploy a WordPress site, fortified with SSL and best security practices such as chroot environments. Additionally, we've taken a deep dive into Docker by deploying a Nextcloud server, demonstrating our ability to manage multiple services on a single server seamlessly.

As you may have noticed in the cases we studied in this chapter, the process of setting up a web application can sometimes be long and complex. But when it comes to the part that concerns NGINX, the configuration is usually pretty simple and straightforward: a couple of directives in a server block, reload the server, and you're done.

Unfortunately, in some cases, while your initial configuration seems to do the trick, you realize over time that your visitors run into a variety of problems or are presented with unexpected error pages. The next chapter will prepare you to face such issues by exploring several leads, should you ever need to troubleshoot your web server.

11
Troubleshooting

Even if you read every single word of this book with the utmost attention, you will not unfortunately be sheltered from all kinds of issues, ranging from simple configuration errors to the occasional unexpected behavior of one module or another. To help you with that, in this chapter, we will attempt to provide solutions for some of the common problems encountered by administrators who are just getting started with NGINX.

This chapter covers the following topics:

- A basic guide containing general tips on NGINX troubleshooting
- How to solve some of the most common install issues
- Dealing with `403 Forbidden` and `400 Bad Request` HTTP errors and more
- Why your configuration does not appear to apply correctly
- A few words about the `if` block behavior

Looking at some general tips on NGINX troubleshooting

Before we begin, whenever you run into some kind of problem with NGINX, you should make sure to follow the recommendations given in the following sections, as they are generally a good source of solutions.

Checking access permissions

A lot of errors that NGINX administrators are faced with are caused by invalid access permissions. On two separate occasions, you have the option to specify a user and group for the NGINX worker processes to run:

- When configuring the build with the `configure` command, you are allowed to specify a user and group that will be used by default (refer to *Chapter 1*).
- In the configuration file, the `user` directive allows you to specify a user and group. This directive overrides the value that you may have defined during the `configure` step.

If NGINX is supposed to access files that do not have the correct permissions, in other words, that cannot be read (and by extension, cannot be written for directories that hold temporary files, for example) by the specified user and group, NGINX will not be able to serve files correctly. Additionally, should your web application encounter an error related to file- or directory-access permissions, the user and group under which your FastCGI or other backend runs should also be investigated.

Testing your configuration

A common mistake is often made by administrators showing a little too much self-confidence: after having modified the configuration file (often without a backup), they reload NGINX to apply the new configuration. If the configuration file contains syntax or semantic errors, the application will refuse to reload. Even worse, if NGINX is stopped (for example, after a complete server reboot) it will refuse to start at all. In either of those cases, remember to follow these recommendations:

- Always keep a backup of your working configuration files in case something goes wrong
- Before reloading or restarting NGINX, test your configuration with a simple command, `nginx -t`, to test your current configuration files, or run `nginx -t -c /path/to/config/file.conf`
- Reload your server instead of restarting it, preferring `systemctl reload nginx` over `systemctl restart nginx` (and `nginx -s reload` instead of `nginx -s stop && nginx`), as it will keep existing connections alive, and thus won't interrupt ongoing file downloads

Have you reloaded the service?

You would be surprised to learn how often this happens: the most complicated situations have the simplest solutions. Before tearing your hair out, before rushing to the forums or IRC asking for help, start with the most simple of verifications.

You just spent two hours creating your virtual host configuration. You've saved the files properly and have fired up your web browser to check the results. But did you remember that one additional step? NGINX, unlike Apache, does not support on-the-fly configuration changes in `.htaccess` files or similar. So take a moment to make sure you have reloaded NGINX with `systemctl reload nginx` or `/usr/local/nginx/sbin/nginx -s reload`, without forgetting to test your configuration beforehand!

Checking logs

There is usually no need to look for the answer to your problems on the internet. Chances are, the answer is already given to you by NGINX in the log files. There are two variations of log files you may want to check. First, check the access logs. These contain information about requests themselves: the request method and URI, the HTTP response code issued by NGINX, and more, depending on the log format you defined:

error.log

```
1 2015/09/12 06:46:19 [error] 10164#0: *221161 open() "/home/example.com/wp-content/plugins/contactformgenerator/documentati
2 2015/09/12 06:46:25 [error] 10164#0: *221162 open() "/home/example.com/wp-content/plugins/seo-interlinking/js/quicksearch.
3 2015/09/12 06:46:25 [error] 10164#0: *221163 open() "/home/example.com/wp-content/plugins/Premium_Gallery_Manager/sprites/
4 2015/09/12 06:46:30 [error] 10164#0: *221164 open() "/home/example.com/wp-content/plugins/rock-form-builder/admin/js/rock-
5 2015/09/12 06:46:30 [error] 10164#0: *221165 open() "/home/example.com/wp-content/plugins/maxbuttons-pro/js/maxbuttons.js"
6 2015/09/12 07:32:23 [error] 10164#0: *221206 open() "/home/example.com/robots.txt" failed (2: No such file or directory),
7 2015/09/12 07:44:08 [error] 10164#0: *221207 open() "/home/example.com/robots.txt" failed (2: No such file or directory),
8 2015/09/12 10:05:14 [error] 10164#0: *221298 open() "/home/example.com/favicon.ico" failed (2: No such file or directory),
```

Figure 11.1: An extract of the Nginx error log for troubleshooting and debugging purposes

More importantly, for troubleshooting, the error log is a goldmine of information. Depending on the level you defined (see the `error_log` and `debug_connection` directives for more details), NGINX will provide details on its inner functioning. For example, you will be able to see the request URI translated to the actual filesystem path. This can be a great help for debugging rewrite rules. The error log should be located in the `/logs/` directory of your NGINX setup, by default `/usr/local/nginx/logs` or `/var/log/nginx`.

In this section, we learned how to troubleshoot NGINX running errors. Moving forward, we'll explore the use of external tools to streamline the aggregation of NGINX errors for more efficient troubleshooting.

Installing a log parser

While NGINX has great logs, at some of the higher levels of logging, they can also be quite exhaustive in the amount of information they log. A good way to not miss information and get a high-level overview of what is going on with NGINX is to install a log parser that can aggregate information and display it in a more approachable format.

> **Note**
>
> One open source tool we can use for this is called **GoAccess**. More info can be found on its website at `https://goaccess.io/`.

The good thing about GoAccess, aside from being free and open source, is that it can be accessed through both the Terminal and your browser. Therefore, it can function as both a monitoring tool that you run in your Terminal and as a reporting tool that generates a kind of dashboard for your stats:

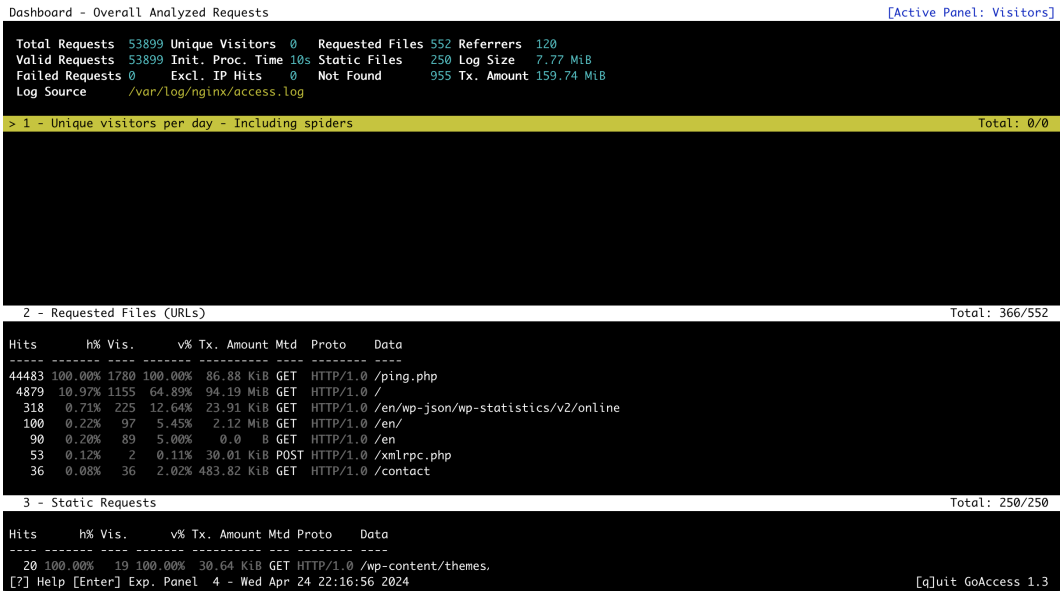


Figure 11.2: GoAccess analyzing and organizing Nginx logs for a clear view

To get started and install GoAccess, you can either check your distribution package manager (whether with `apt`, `dnf` or else) or download and compile it manually:

```
wget http://tar.goaccess.io/goaccess-1.9.1.tar.gz
tar -xzvf goaccess-1.9.1.tar.gz
cd goaccess-1.9.1/
./configure --enable-utf8 --enable-geoip=mmdb
make
make install
```

Once installed, using it is very straightforward; you can get a Terminal view by running the following command:

```
goaccess /var/log/nginx/access.log --log-format=COMBINED
```

Here, the `COMBINED` log format refers to the default log format of NGINX, but it is also compatible with Apache HTTPd logs. If you want to get an HTML report that you can view in a browser, or perhaps email to someone for reporting, then run the following:

```
goaccess /var/log/nginx/access.log -o report.html
--log-format=COMBINED
```

A neat feature of GoAccess is that it can also provide a real-time auto-updating HTML page by using the `--real-time-html` flag. Enabling this will add some WebSocket code to the report that will fetch the latest stats continuously. Create a location block for your report and point it to your `report.html` to have a report always available online.

We've covered how to handle NGINX log errors. Up next, we'll dive into the challenges of compiling and installing NGINX.

Troubleshooting install issues

There are typically four sources of errors when attempting to install NGINX or to run it for the first time:

- Some of the prerequisites are missing or an invalid path to the source was specified. More details about prerequisites can be found in *Chapter 1*.
- After having installed NGINX correctly, you cannot use the SSL-related directives to host a secure website. Have you made sure to include the SSL module correctly during the `configure` step? More details are in *Chapter 1*.
- NGINX refuses to start and outputs a message similar to `[emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)`. This error signifies that another application is utilizing the network port `80`. This could either mean that another web server, such as Apache, is already running on the machine, or that you don't have the proper permissions to open a server socket on this port. This can happen if you are running NGINX from an underprivileged system account.

NGINX refuses to start and outputs a message similar to `[emerg] 3629#0: open() "/path/to/logs/access.log" failed (2: No such file or directory)`. In this case, one of the files that NGINX tries to open, such as a log file, cannot be accessed. This could be caused by invalid access permissions or by an invalid directory path (for example, when specifying log files to be stored in a directory that does not exist on the system). After addressing NGINX's compilation challenges, we'll explore location block and error nuances. Though NGINX functions, we'll learn to adjust these settings to meet our expectations.

Looking at the 403 forbidden custom error page

If you decide to use `allow` and `deny` directives to allow or deny access, respectively, to a resource on your server, clients who are being denied access will usually fall back on a `403 Forbidden` error page. Imagine you have carefully set up a custom, user-friendly 403 error page for your clients to understand why they are denied access. Unfortunately, you cannot get that custom page to work, and clients still get the default NGINX 403 error page:

```
server {
    [...]
    allow 192.168.0.0/16;
    deny all;
    error_page 403 /error403.html;
}
```

The problem is simple: NGINX also denies access to your custom 403 error page! In such a case, you need to override the access rules in a `location` block specifically matching your page. You can use the following code to allow access to your custom 403 error page only:

```
server {
    [...]
    location / {
        error_page 403 /error403.html;
        allow 192.168.0.0/16;
        deny all;
    }
    location = /error403.html {
        allow all;
    }
}
```

If you are going to have more than just one error page, you could specify a `location` block matching all error page filenames:

```
server {
    [...]
    location / {
        error_page 403 /error403.html;
        error_page 404 /error404.html;
        allow 192.168.0.0/16;
        deny all;
    }
```

```
    location ~ "^/error[0-9]{3}.html$" {
        allow all;
    }
}
```

All your visitors are now allowed to view your custom error pages.

Exploring 400 Bad Request

Occasionally, you may run into a recurring issue with some of your websites: NGINX returns `400 Bad Request` error pages to random visitors, and this only stops happening when visitors clear their cache and cookies. The error is caused by an overly large header field sent by the client. Most of the time, this is when cookie data exceeds a certain size. In order to prevent further trouble, you can simply increase the value of the `large_client_header_buffers` directive in order to allow a larger cookie data size:

```
large_client_header_buffers 4 16k;
```

In addition to adjusting the `large_client_header_buffers` for resolving 400 Bad Request errors, it's also worth examining directives such as `proxy_buffers`, `proxy_buffer_size`, and `proxy_busy_buffers_size`. Modifying these settings can further mitigate error 400 by optimizing how NGINX handles incoming data.

Looking at truncated or invalid FastCGI responses

When setting up an NGINX frontend for a website that heavily relies on **AJAX** (short for **Asynchronous JavaScript and XML**), along with a FastCGI backend such as PHP, you may run into different sorts of problems. If your server returns truncated AJAX responses, invalid JSON values, or even empty responses, you may want to check your configuration for the following elements:

- Have you set up a writable directory for FastCGI temporary files? Make sure to do so via the `fastcgi_temp_path` directive.
- If `fastcgi_buffering` is set to `off`, all FastCGI responses are forwarded to the client synchronously, in chunks of a certain size (determined by `fastcgi_buffer_size`).
- In some cases, increasing the size and number of buffers allocated to storing FastCGI responses prevents responses from getting truncated. For example, use `fastcgi_buffers 256 8k;` for 256 buffers of 8 kilobytes each.

Exploring location block priorities

This problem frequently occurs when using multiple location blocks in the same server block: configuration does not apply as you thought it would.

As an example, suppose that you want to define a behavior to be applied to all image files that are requested by clients:

```
location ~* .(gif|jpg|jpeg|png)$ {
    # matches any request for GIF/JPG/JPEG/PNG files
    proxy_pass http://imageserver; # proxy pass to backend
}
```

Later on, you decide to enable automatic indexing of the `/images/` directory. Therefore, you decide to create a new `location` block, matching all requests starting with `/images/`:

```
location ^~ /images/ {
    # matches any request that starts with /images/
    autoindex on;
}
```

With this configuration, when a client requests to download `/images/square.gif`, NGINX will apply the second location's block only. Why not the first one? The reason is that `location` blocks are processed in a specific order. For more information about `location` block priorities, refer to the *Location block* section in *Chapter 3*.

Looking at if block issues

In some situations, if not most, you should avoid using `if` blocks. There are two main issues that occur, regardless of the NGINX build you are using.

Inefficient statements

There are some cases where `if` is used inappropriately, in a way that risks saturating your storage device with useless checks:

```
location / {
    # Redirect to index.php if the requested file is not found
    if (!-e $request_filename) {
       rewrite ^ index.php last;
    }
}
```

With such a configuration, every single request received by NGINX will trigger a complete verification of the directory tree for the requested filename, thus requiring multiple storage disk access system calls. If you test `/usr/local/nginx/html/hello.html`, NGINX will check `/`, `/usr`, `/usr/local`, `/usr/local/nginx`, and so on. In any case, you should avoid resorting to such a statement, for example, by filtering the file type beforehand (which could be done by making such a check only if the requested file matches specific extensions):

```
location / {
    # Filter file extension first
    if ($request_filename !~ ".(gif|jpg|jpeg|png)" {
       break;
    }
    if (!-f $request_filename) {
       rewrite ^ index.php last;
    }
}
```

Unexpected behavior

The `if` block should ideally be employed in simple situations, as its behavior might be surprising in some cases. Apart from the fact that `if` statements cannot be nested, the following situations may present issues:

```
# Two consecutive statements with the same condition:
location / {
    if ($uri = "/test.html") {
       add_header X-Test-1 1;
       expires 7;
    }
    if ($uri = "/test.html") {
       add_header X-Test-1 1;
    }
}
```

In this case, the first `if` block is ignored and only the second one is processed. However, if you insert a *Rewrite module* directive in the first block, such as `rewrite`, `break`, or `return`, the first block will be processed and the second one will be ignored.

There are many other cases where the use of `if` causes problems:

- Having `try_files` and `if` statements in the same location block is not recommended, as the `try_files` directive will, in most cases, be ignored.
- Some directives are theoretically allowed within the `if` block, but can create serious issues; for instance, `proxy_pass` and `fastcgi_pass`. You should keep those within `location` blocks.
- You should avoid using `if` blocks within a `location` block that captures regular expression patterns from within its modifier.

These issues originate from the fact that while the NGINX configuration is written in what appears to be a declarative language, directives from the Rewrite module, such as `if`, `rewrite`, `return`, or `break`, make it look like event-based programming. In general, you should try to avoid using directives from other modules within `if` blocks as much as possible.

Summary

Most of the problems you can run into will occur during the early configuration stages while you test your server before production. These problems are usually easier to deal with because you are mentally prepared for the challenge, and more importantly, because NGINX points out syntax or configuration errors on startup. It is, on the other hand, much more difficult to identify the cause of malfunctions while your websites are actually in production. But once again, NGINX saves the day: if you properly configure log files (both access and error logs) and adopt the habit of reading them regularly, you will find that problem-solving is made easy.

This concludes our journey with NGINX, throughout which we have walked through a large number of subjects, from the basic mechanisms of the HTTP server to web application deployment and troubleshooting.

If you want to know more about NGINX, we invite you to dive into the vibrant NGINX community, a collaborative hub teeming with expertise and innovative ideas. Whether seeking advice for your projects or offering your own insights, the community is an invaluable resource for broadening your understanding and refining your skills. Engage in discussions, share solutions, and connect with fellow enthusiasts to further your journey with NGINX.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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 packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@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.

Other Books You May Enjoy

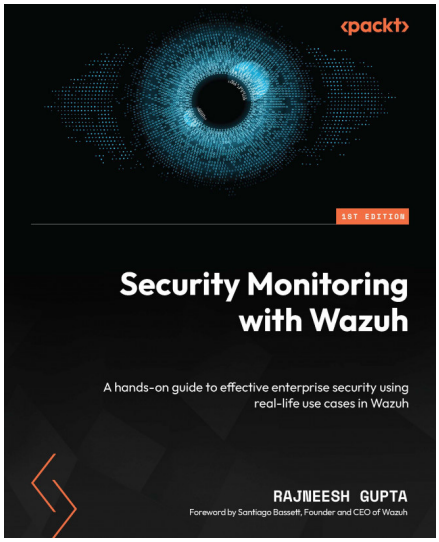
If you enjoyed this book, you may be interested in these other books by Packt:

Pentesting Active Directory and Windows-based Infrastructure

Denis Isakov

ISBN: 978-1-80461-136-4

- Understand and adopt the Microsoft infrastructure kill chain methodology
- Attack Windows services, such as Active Directory, Exchange, WSUS, SCCM, AD CS, and SQL Server
- Disappear from the defender's eyesight by tampering with defensive capabilities
- Upskill yourself in offensive OpSec to stay under the radar
- Find out how to detect adversary activities in your Windows environment
- Get to grips with the steps needed to remediate misconfigurations
- Prepare yourself for real-life scenarios by getting hands-on experience with exercises



Security Monitoring with Wazuh

Rajneesh Gupta

ISBN: 978-1-83763-215-2

- Find out how to set up an intrusion detection system with Wazuh
- Get to grips with setting up a file integrity monitoring system
- Deploy Malware Information Sharing Platform (MISP) for threat intelligence automation to detect indicators of compromise (IOCs)
- Explore ways to integrate Shuffle, TheHive, and Cortex to set up security automation
- Apply Wazuh and other open source tools to address your organization's specific needs
- Integrate Osquery with Wazuh to conduct threat hunting

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *NGINX HTTP Server*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835469873`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly