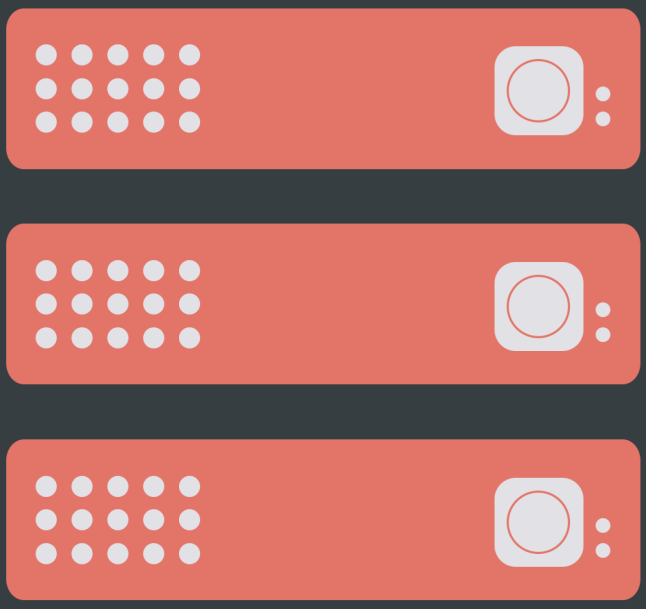


Servers for Hackers

Chris Fidao

Servers for Hackers

Server Administration for Programmers

Chris Fidao

This book is for sale at http://leanpub.com/serversforhackers

This version was published on 2015-05-01

Tweet This Book!

Please help Chris Fidao by spreading the word about this book on Twitter!

The suggested hashtag for this book is #srvrsforhackers.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#srvrsforhackers

Contents

Servers . . . . . **i**
Video Site i
Book Issues ii

Introduction . . . . . **iii**

Accidental Sysadmin Syndrome . . . . . **iv**
Assumptions iv

Linux Distributions . . . . . **v**

The Sandbox **1**
Install Virtualbox and Vagrant 1
Configure Vagrant 2
Vagrant Up! 6
Basic Commands 10
Basic Software 11
Review 13

Security **14**

Users and Access **15**
IP Addresses 15
Creating a New User 15
Making Our User a Super User 16

Setting Up the Firewall: Iptables **27**
Adding these rules 28
Inserting Rules 31
Deleting Rules 31
Saving Firewall Rules 33
Defaulting to DROP Over ACCEPT 34
Logging Dropped Packets 35

Fail2Ban **37**
Iptables Integration 37
Installation 38
Configuration 39

Automatic Security Updates **43**

Package Managers 45

Apt **46**
Installing 46
Repositories 47
Examples 47
Searching Packages 50

Permissions and User Management 52

Permissions **53**
Checking Permissions 53
Changing Permissions 54

User Management **57**
Creating Users 59
Umask & Group ID Bit 60
Running Processes 63

Webservers 64

HTTP, Web Servers and Web Sites **65**
A Quick Note on DNS 66

DNS & Hosts File **68**
Xip.io 70
Virtual Hosts 71

Hosting Web Applications **75**
Three Actors 75

Apache **83**
Installing 83
Configuration 84
Virtual Hosts 87
Apache and Web Applications 91

MPM Configuration . . . 114
Security Configuration . . . 117
Envvars . . . 118

Nginx . . . 122
Features . . . 122
Installation . . . 123
Web Server Configuration . . . 124
Servers (virtual hosts) . . . 126
Integration with Web Applications . . . 133

PHP . . . 143
Installation . . . 143
Configuration . . . 144
PHP-FPM . . . 146

Server Setup for Multi-Tenancy Apps . . . 157
DNS . . . 157
Multi-Tenancy in Apache . . . 159
Multi-Tenancy in Nginx . . . 160

SSL Certificates . . . 162

SSL Overview . . . 163
Using SSL in Your Application . . . 163

Creating Self-Signed Certificates . . . 165
Creating a Wildcard Self-Signed Certificate . . . 167
Apache Setup . . . 169
Nginx Setup . . . 173
One Server Block . . . 174

Extra SSL Tricks . . . 176

Multi-Server Environments . . . 177

Implications of Multi-Server Environments . . . 178
Asset Management . . . 178
Sessions . . . 179
Lost Client Information . . . 180
SSL Traffic . . . 181
Logs . . . 182

Load Balancing with Nginx . . . 184

Balancing Algorithms . . . 184
Configuration . . . 185

Load Balancing with HAProxy . . . 192
Common Setups . . . 192
Installation . . . 192
HAProxy Configuration . . . 193
Monitoring HAProxy . . . 200
Sample NodeJS Web Server . . . 201

SSL with HAProxy . . . 203
HAProxy with SSL Termination . . . 203
HAProxy with SSL Pass-Through . . . 206
Sample NodeJS Web Server . . . 208

Web Cache . . . 210

Nuts and Bolts of HTTP Caching . . . 211
Object Caches . . . 211
Web Caches . . . 211
Types of HTTP Caches . . . 212

An Origin Server . . . 214
Testing Caching Mechanisms . . . 214

Nginx Web Caching . . . 219
Use Cases . . . 219
How It Will Work . . . 220
Origin Server . . . 220
Cache Server . . . 221
Proxy Caching . . . 226
Example: Caching Specific URIs . . . 229

Varnish . . . 231
Origin Server . . . 231
Install Varnish . . . 232
Basic Configuration . . . 232
Increasing Cache Hit Rate . . . 238
Varnish Tools . . . 241
Extra Resources . . . 244

Logs **245**

Logrotate **246**
What does Logrotate do? 246
Configuring Logrotate 246
Going Further 252

Rsyslog **254**
Configuration 254
Usage 258
Should I Use Rsyslog? 262
Sending To Rsyslog From An Application 262

File Management, Deployment & Configuration Management **263**

Managing Files **264**
Copying Files Locally 264
SCP: Secure Copy 264
Rsync: Sync Files Across Hosts 265
Deployment 268

Auto-deploy with GitHub **269**
How it Works 269
Node Listener 269
Shell Script 271
Putting it together 272
Firewall 272

Configuration Management with Ansible **274**
Install 274
Managing Servers 275
Basic: Running Commands 276
Basic Playbook 278
Roles 282
Facts 291
Vault 293

SSH **298**

Logging in **299**

SSH Config **300**

SSH Tunneling **302**

Local Port Forwarding . . . 302
Remote Port Forwarding . . . 304

One-Off Commands & Multiple Servers . . . 306
Basic Ansible . . . 306

Monitoring Processes . . . 309

A Sample Script . . . 310

System Services . . . 311
System V Init (SysVinit, SysV) . . . 311
Upstart . . . 312
The Service Command . . . 313
Systemd . . . 314
Using These Systems . . . 315

Supervisord . . . 317
A Chain of Process Monitors . . . 317
Installation . . . 318
Configuration . . . 318
Controlling Processes . . . 320
Web Interface . . . 322

Forever . . . 323
Installation . . . 323
Usage . . . 323

Circus . . . 325
Installation . . . 325
Configuration . . . 326
Controlling Processes . . . 328
Web Interface . . . 330
Starting on Boot . . . 331

Development and Servers . . . 333

Serving Static Content . . . 334
Built-In . . . 334
NodeJS . . . 334
Dynamic Content . . . 336

Servers

Servers can be fun!

Knowing how to setup, run and administer a server can be as empowering as coding itself!

Some application have needs stretching beyond what hosting providers can give. This shouldn't stop us from building the application.

Servers can be hard!

Consumers expect and demand services to be functioning. Downtime can cost real money, and is often met with frustration and anger.

At the same time, servers are increasingly commodified. Hosting once involved a few, powerful servers. Now, the modern "Cloud" consists of many small, cheap virtual machines. Virtual machines commonly die for many reasons.

The end result is that we need to build for failure. This is a Hard Problem™, and requires us to know a lot about the servers running our applications.

This book exists because we developers are now faced with System Administration issues. We need to at least know the basics of what goes into hosting and serving our application!

So, let's not get stuck with limiting hosting or a broken server!

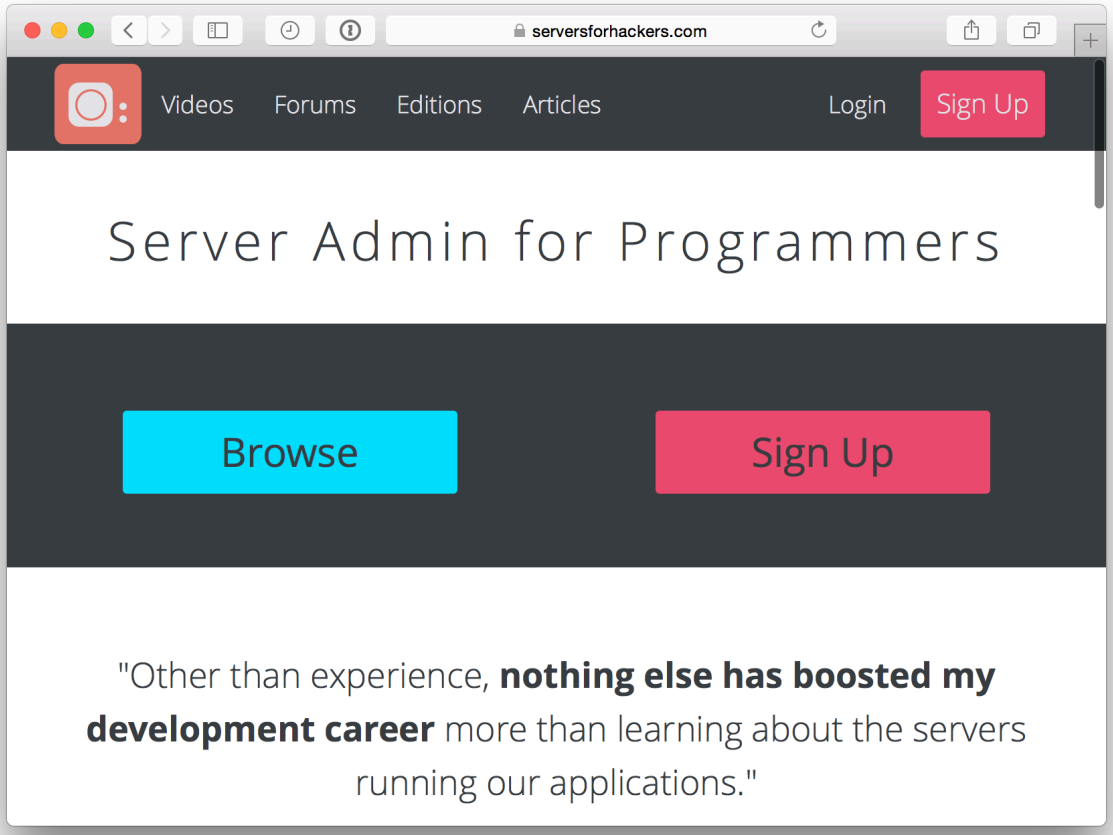
Video Site

Since publishing this book, I've also started a subcription-based video site, found at https://serversforhackers.com[1]. This includes all book videos, and many, many more.

I'll be adding new videos weekly! These continue to concentrate on topics important to web servers and web development, from the basic to the complex.

The videos all come with a write-up of the commands and information presented in the video, usually along with some extra resources. This makes the videos easy to come back to for quick reference later.

[1] https://serversforhackers.com



Servers for Hackers Video Site

Book Issues

All feedback is hugely appreciated! Any questions, comments, issues, stories of glory/valor and praise can be directed to the Github repository[2] created for feedback!

https://github.com/Servers-for-Hackers/the-book

[2] https://github.com/Servers-for-Hackers/the-book

Introduction

Accidental Sysadmin Syndrome

You're a developer.

A server broke, and you're the only one around to fix it.

You have a special-needs application that requires specific software.

You need to setup a development server, and will spend half of your day trying to get some "simple" configuration to work.

These are symptoms of Accidental Sysadmin Syndrome.

This book is for developers who find themselves needing or wanting to be a SysAdmin.

Assumptions

This book assumes at least a passing familiarity with the command line. Those who have logged into the shell and poked around a server before will benefit the most.

If you are new to the command line, concentrate on getting comfortable with Vagrant. This will help familiarize you with using the command line and servers.

Linux Distributions

There are many distributions of Linux. Some popular ones are Arch, Debian, Ubuntu, Redhat, CentOS, Fedora and CoreOS.

Many of these distributions are related to each other in some way. For example, some of these distributions are "downstream" from others.

A downstream Linux distribution includes the upstream's distribution's changes, and may add their own.

For example, Ubuntu is based on Debian and is considered downstream of Debian. CentOS is based on RedHat and is therefore downstream from RedHat. RedHat sponsors Fedora and so Fedora is very similar to RedHat and CentOS (although it has a much more rapid release cycle).

Each distribution has opinions about Linux and its use. **It would be too cumbersome to cover all topics for each distribution and so I've chosen to concentrate on Ubuntu**.

This book concentrates on Debian/Ubuntu, however, the serversforhackers.com[3] video site covers multiple distributions!

Ubuntu is one of the most popular server and desktop distributions. It has a great set of configurations that come out of the box, allowing us to worry less about configuration until we need to. This makes it easy to use.

Ubuntu updates software packages quickly relative to some other distributions. However, updating to the latest software makes it easier to introduce new bugs and version conflicts.

Luckily, Ubuntu's LTS releases are a good solution to this potential issue.

LTS stands for Long Term Support

LTS versions are released every 2 years but support for them last 5 years. This makes them ideal for longer-term use.

As major versions are released yearly, only every *other* major release of Ubuntu is an LTS. The current LTS is 14.04 - the next LTS release will be 16.04.

Trusty, the codename for Ubuntu 14.04, was released in April of 2014. This will be a relevant server for at least 2 years.

[3] https://serversforhackers.com

LTS releases offer more stability and security, and do not prevent us from installing the latest software when we need to. This makes them ideal candidates for every-day server usage.

Popularity is Relative

RedHat Enterprise (RHEL) is a popular distribution in the enterprise world. Many hosting companies use CentOS along with cPanel/WHM or Plesk control panels. In the open source/startup worlds Ubuntu is one of the most popular distributions of Linux.

Because Ubuntu is closely tied to Debian, most topics included here will be exactly the same for Debian. Some topics may vary slightly.

For RedHat/CentOS distributions, most topics will have small-to-large differences from what you read here.

In any case, much of what you learn here will be applicable to all distributions. The difference in distributions is usually just configuration.

I recommend this Rackspace knowledge-base article for more information on the various Linux distributions: http://www.rackspace.com/knowledge_center/article/choosing-a-linux-distribution[4].

[4] http://www.rackspace.com/knowledge_center/article/choosing-a-linux-distribution

The Sandbox

If you want a sandbox - a place to safely play with a server - this chapter is for you.

The topics of the "Sandbox" section is **not** necessary to follow along in this book, but it will be helpful.

You'll learn how to setup a local server on which you can develop an application or experiment with new technology. As a bonus, you'll avoid mucking up your computer with development software!

We'll briefly cover using Vagrant to setup a local server.

The benefit of Vagrant is that it will let us use a "real" server to test on. You can create a server also used in production. Virtual servers are also safe - we can *thoroughly* mess them up, throw them away and recreate them as much as we need.

Let's get started with Vagrant!

Install Virtualbox and Vagrant

Virtualbox is a tool for creating Virtual Machines. Vagrant is a tool that lets you easily create and manage virtual machines.

Vagrant takes care of file sharing, network setup and other sticky topics.

A Virtual Machine is a (guest) computer running inside of your (host) computer. VirtualBox "virtualizes" hardware by making virtual servers think they are running on real hardware.

A guest computer can be almost anything - Windows, Mac, Linux or other operating systems.

Here's some important vocabulary: Your computer is called the "**host**" machine. Any virtual machine running within the host machine is called a "**guest**" machine.

I'll use the term "virtual machine" with "server" interchangeably, as we'll be creating Ubuntu servers (VMs) to learn on.

To get started, the first step is to install Virtualbox and Vagrant. These are available for Windows, Mac and Linux. Intalling them only involves browsing to their websites and downloading/running their installers. You may need to restart your Windows after installing Vagrant.

For this book, you will need Vagrant version 1.5 or higher. Most versions of Virtualbox should work, I always update to the latest of these two tools.

Configure Vagrant

Once you have installed these, we can get started! We'll get Vagrant going by running the following commands on our host machine.

On Mac, open up the Terminal app. On Windows, you can use the CMD prompt (or your command line tool of choice) to run Vagrant commands.

On Mac:

```
mkdir -p ~/Sites/sfh
cd ~/Sites/sfh
vagrant init ubuntu/trusty64
```

On Windows:

```
mkdir C:\sfh
cd C:\sfh
vagrant init ubuntu/trusty64
```

From here on, I won't differentiate between Windows and Mac commands. We'll mostly be within a server in any case, so the commands will not vary no matter what type of computer your host is.

The `vagrant init` command creates a new file called `Vagantfile`. This file is configured to use Ubuntu 14.04 LTS server, codenamed "trusty". This is the server we'll be using for this book.

The Vagrantfile created will look something like this (when all the comments are stripped out):

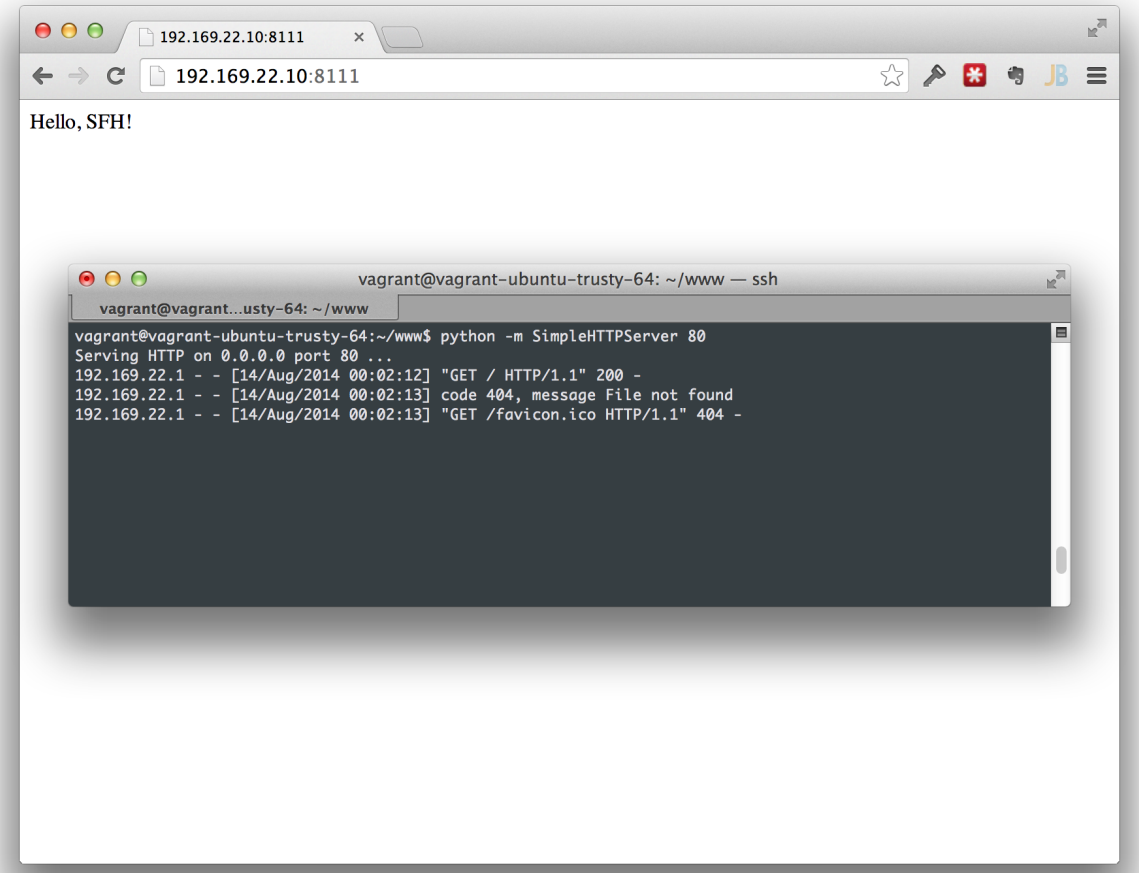
File: Vagrantfile

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

    config.vm.box = "ubuntu/trusty64"

end
```

If you look at your file, you'll see lots of comments, which show some configurations you can use. I'll cover a few that you should know about.

Networking

The basic install of Vagrant will do some "port forwarding". For example, if Vagrant forwards port 8080 to the server's port 80, then we'll go to http://localhost:8080 in your browser to reach the server's web server at port 80. This has some side effects.



A side effect of this port forwarding has to do with interacting with web applications. You'll need to access web pages in your browser using the port which Vagrant sets up, often "8888". Instead of "http://localhost", you'll use "http://localhost:8888" in the browser. However, your application may not be coded to be *aware* of the non-standard port (8888). The application may redirect to, create links for or submit forms to standard port 80 instead of the forwarded port!

I like to get around this potential problem by assigning an private-network IP address to my Vagrant server.

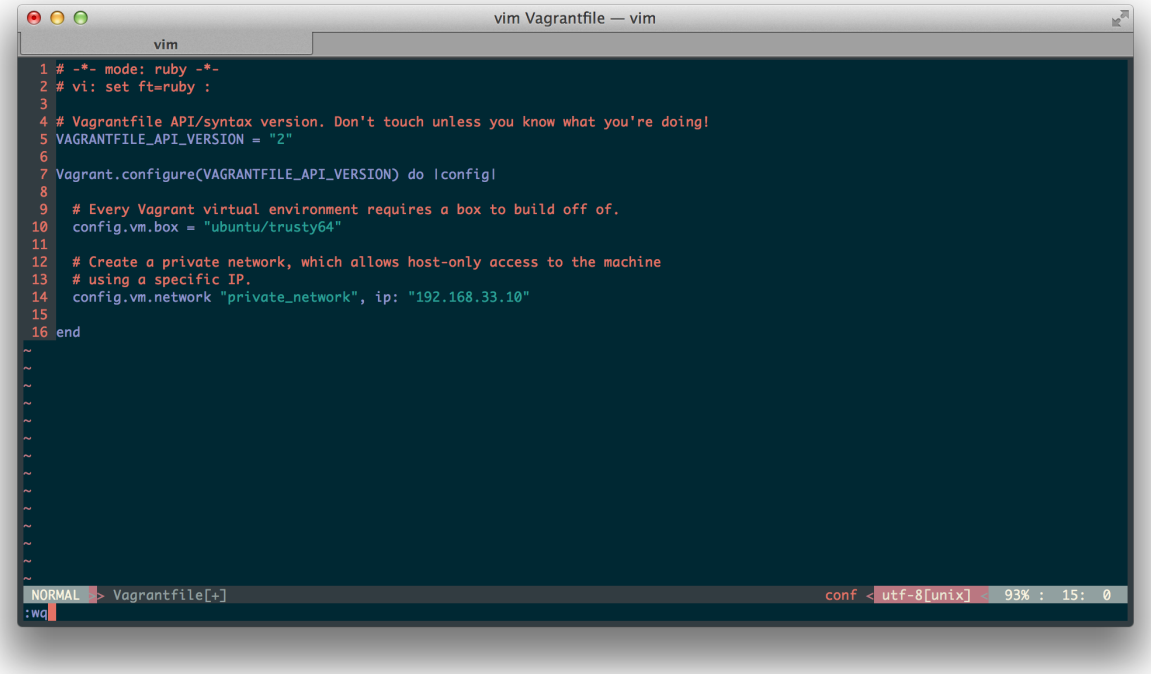
To do this, open up your `Vagrantfile` and make it look like this:

File: Vagrantfile

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

    config.vm.box = "ubuntu/trusty64"

    config.vm.network :private_network, ip: "192.168.22.10"

end
```

The `private_network` directive tells Vagrant to setup a private network. Our host and guest machines can communicate on this network. This assigns the guest server the IP address of 192.168.22.10. Note that each server should have a unique IP address just in case they are run at the same time.

There are IP address ranges set aside for private networks. Generally you can use `10.0.0.0 - 10.255.255.255`, `172.16.0.0 - 172.31.255.255`, and `192.168.0.0 - 192.168.255.255`. However, always avoid the lower and upper IP addresses within those ranges, as they are often reserved.



Vagrant Up!

Once the `Vagrantfile` changes are saved, we can run the `vagrant up` command. This will download the `ubuntu/trusty64` base server ("box") and run it with our set configuration.

```
vagrant up
```

If Vagrant cannot find the Vagrantfile, you need to `cd` into the directory containing the Vagrantfile.

You'll see some output as Vagrant sets up the Ubuntu server. Once it's complete, run `vagrant status` to see that it's powered on and running.

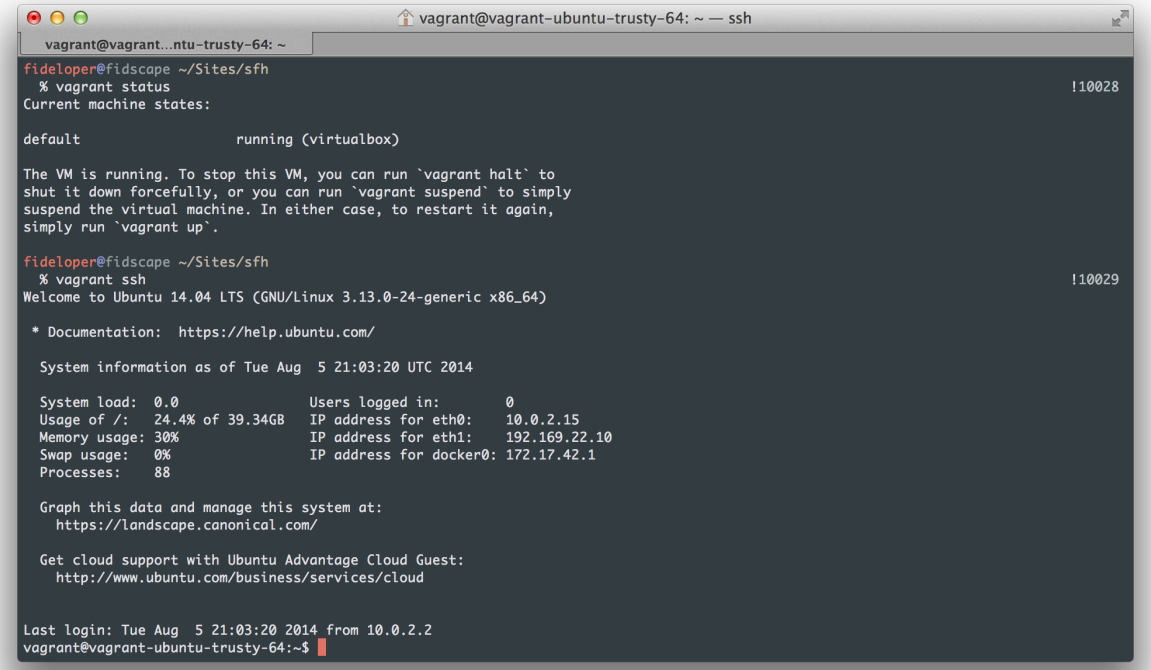
```
vagrant status
```

You should see output similar to this:

```
Current machine states:

default                   running (virtualbox)

The VM is running. To stop this VM, you can run `vagrant halt` to
shut it down forcefully, or you can run `vagrant suspend` to simply
suspend the virtual machine. In either case, to restart it again,
 run `vagrant up`.
```

Our machine, named "default" is running, using VirtualBox.

Now we need to log into this server. Vagrant sets up a way to log in without needing a password nor SSH key. Run `vagrant ssh` to log into the server!



Congratulations, you're now inside of a real server! Poke around a bit - try some of these commands out if they are not familiar to you:

- `ll` - A buit-in alias for the command `ls -alF`, this will list all files within the current directory
- `lsb_release -a` - A command to show all release information about this server
- `top` - A command to show running processes and various system information. Use the `ctrl+c` keyboard shortcut to return to the prompt.
- `clear` - A command to clear currently visible output within your terminal
- `df -h` - See how much hard drive space is used/available

File Sharing

Vagrant sets up file sharing for you. The default shares the server's `/vagrant` directory to the host's directory containing the Vagrantfile.

In our example, the host machine's `~/Sites/sfh` directory is shared with the guest's `/vagrant` directory.

> The tilde ~ expands to the current user's home directory. `~/Sites/sfh` expands to `/Users/fideloper/Sites/sfh`.

List the contents of the `/vagrant` directory within your server:

```
ls -la /vagrant
```

Its output will be something like this:

```
drwxr-xr-x  1 vagrant vagrant  136 Jun 14 16:56 ./
drwxr-xr-x 23 root    root    4096 Jun 14 19:33 ../
drwxr-xr-x  1 vagrant vagrant  102 Jun 14 16:54 .vagrant/
-rw-r--r--  1 vagrant vagrant  480 Jun 14 16:56 Vagrantfile
```

We see our `Vagrantfile` and a hidden `.vagrant` directory containing some meta data used by Vagrant.

On my host machine, I'll create a new text file in `~/Sites/sfh` named `hello.txt`:

```
echo "Hello World" > ~/Sites/sfh/hello.txt
```

Now if I log into the guest server, I'll see that file is available there as well:

```
# See files in /vagrant
cd /vagrant
ls -la

# Output the content of "hello.txt"
# with the "cat" command
cat /vagrant/hello.txt // Output: "Hello World"
```

This allows us to edit files from our host machine while running the server software within our guest server!

A Vagrantfile with the default file sharing configuration in place would look like this:

File: Vagrantfile

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

    config.vm.box = "ubuntu/trusty64"

    config.vm.network :private_network, ip: "192.168.22.10"

    # Share Vagrantfile's directory on the host with /vagrant on the guest
    config.vm.synced_folder ".", "/vagrant"

end
```

Server Network

Let's check out the network configuration. Within the server, run the command `ifconfig`:

```
ifconfig
```

This usually has a good amount of output:

```
eth0      Link encap:Ethernet  HWaddr 08:00:27:aa:0e:10
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:feaa:e10/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:558 errors:0 dropped:0 overruns:0 frame:0
          TX packets:379 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:56936 (56.9 KB)  TX bytes:48491 (48.4 KB)

eth1      Link encap:Ethernet  HWaddr 08:00:27:ac:ef:d2
          inet addr:192.168.22.10  Bcast:192.168.22.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1188 (1.1 KB)  TX bytes:958 (958.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:888 (888.0 B)  TX bytes:888 (888.0 B)
```

The `ifconfig` command will one day be replaced by the `ip` command, but not yet!

The `ifconfig` command output a lot of content! What are we looking at? Well without getting too deep into this, we are looking at three networks setup within this server. Each network is called an "interface".

- `lo` - The **lo**opback interface. This is used for internal communication between services within the server. This is "localhost" - 127.0.0.1
- `eth0` and `eth1` - These are two additional networks created as well. We can see the IP address we assigend the server at `eth1` - 192.168.22.10. The server also has its own private network, with the IP address 10.0.2.15 assigned to this machine.

The `ifconfig` command is a quick way to check the IP address of your server as well as see the various networks the server is connected to.

You'll *always* see a loopback interface. You'll *usually* see an internal network, useful for communicating within a local network such as a data center.

Most server providers will assign a server a public IP address. Servers with a public IP address can usually be reached by users on the internet.

Basic Commands

We'll be using the command line for 99.9% of this book. On Mac and most Linux desktop distributions, this means using the Terminal app.

On Windows, this means using the CMD prompt, or any other shell you might install. I'm personally partial to Git Bash, which is usually installed alongside Git on Windows. You can run the most common Linux commands with it.

If you're not logged into your Vagrant server, log back in using `vagrant ssh`.

Here are some commands you'll need to know for getting around a server:

`pwd` - Print working directory. The "working directory" is the directory you are current in. When you first log into a server, you're usually placed in the user's "home" directory, most often at `/home/username`. In our Vagrant server, we'll be placed in the `/home/vagrant` directory when we log in.

`ls` - List Directory Contents

```
# List contents of current working directory
ls

# List contents in a list form, with extra information:
ls -l

# List contents, including "hidden" files/folders
ls -la

# Add human-readable file/folder sizes:
ls -lah
```

`cd` - Change Directory.

```
# Change into the "/home/fideloper/sites/sfh" directory.
cd /home/fideloper/sites/sfh

# Same as above, but with the "~" shortcut
# to the current users home directory
cd ~/sites/sfh
```

`mkdir` - Create a directory

```
# Create the `sfh` directory
# inside of /home/fideloper/sites/sfh
mkdir ~/sites/sfh

# Create the /home/fideloper/sites/sfh directory and
# any directory in between that doesn't exist
mkdir -p ~/sites/sfh
```

`rm` - Delete a file or directory

```
# Delete (permanently) the `file.ext` file.
rm /path/to/file.ext

# Delete (recursively) the `/path/to/directory` directory.
rm -r /path/to/directory

# the additional `f` flag is to "force" the action,
# without prompting to make sure you want to do it.
# This is dangerous.
rm -rf /path/to/directory
```

Basic Software

When we get a new server, it is useful to install basic tools that are used again and again. What tools you install will change as dictated by your requirements and experience.

These can include editors (vim, nano), http tools (wget, curl), monitoring tool (htop), searching tools (ack) and anything else! Your personal list of favorites will grow as you gain experience.

Here's what I install:

- **curl** - Making HTTP requests
- **wget** - Retrieve files from the web
- **unzip** - Unzip zip files
- **git** - Git version control
- **ack** - An advanced search tool for searching content of files
- **htop** - Interactive process viewer (better than the simple "top")
- **vim** - The timeless editor. Pro-Tip: Hit "esc" then type ":q" then hit "Enter" to quit. Now you know.
- **tmux** - Terminal Multiplexor - Basically, split your terminal session into different panes
- **software-properties-common** - This is specific to Ubuntu. We'll use it to add software repositories that allow us to install the latest software.

As of Ubuntu 14.04, the `add-apt-repository` command is now included in the `software-properties-common` package rather than the `python-software-properties` package.

Don't worry if you're not sure what that means or aren't familiar with the `add-apt-repository` command, it will be covered in the "Package Managers" section.

Install the Basics

The first thing we'll use is the `apt-get` command to install our packages:

```
1 sudo apt-get install curl wget unzip git ack-grep htop \
2 vim tmux software-properties-common
```

Let's cover this command:

- **sudo** - We used "sudo" as we need to install these items as a super user (the root user, essentially). Only some users are allowed to use "sudo". If you are already user "root" while installing these, then you don't need to use "sudo" before any command.
- **apt-get install** - Install packages with APT
- Then we list all the packages we want to install, separated by a space

This command will prompt us to make sure we want to install all of these packages. We could also add the `-y` flag to skip the prompt: `sudo apt-get install -y curl wget [...]`.

Note that I also split the command into two lines by escaping the newline character with a backslash.

Review

This chapter was a quick primer on Vagrant and some server basics. We saw how to download and install VirtualBox and Vagrant, how to configure Vagrant, and how to install basic software.

This is not the only way to go about this. You may want to use a remote server, rather than have one running on your local computer. I suggest using Digital Ocean or Linode. Use what works best for you!

Security

When you create a server that is publicly accessible, the first thing you should do is guard the server against unwanted intrusion.

The following chapters will show you measures you should take to lock down any server.

Users and Access

Some security precautions are always warranted when we get a fresh server. This is *especially* important if the server is open to a public network.

The servers spun up by providers are usually open to the public. Providers assign the servers a IP address on a public network upon creation.

IP Addresses

Freshly provisioned servers aren't safe just because they haven't announced their presence.

Providers purchase IP addresses in blocks. Finding the ranges of IP addresses used by a hosting provider is not difficult. Providers have only a limited number of public-facing IP addresses they can assign.

As a result, IP addresses are often released and reassigned when customers destroy and create servers.

This means that someone likely knows the IP address of your server. Automated bots may come snooping to see what vulnerabilities might be open on your server the instant it's created!

Compounding this, many providers provide the root user's password in plaintext within email.

From a security point of view, none of the above is particularly great. We need to lock new servers down with some basic security.

In the following chapter, we'll address these concerns and more. Here's what we'll cover:

1. Creating a new (non-root) user
2. Allowing this user to use "sudo" for administrative privileges
3. Stopping user "root" from remotely logging in via SSH
4. Configuring SSH to change the port and add other restrictions
5. Creating an SSH key on our local computer for logging in as our new user
6. Turning off password-based authentication, so we *must* use an SSH key to access the server

Creating a New User

Let's create a new user. First, of course, you need to log into your server. Within Vagrant, this is simply the command `vagrant ssh`.

If, however, you're using one of the many cloud (or traditional) providers, then you need to SSH in using the usual means:

```
ssh username@your-server-host
```

The “username” is the username provided by you, and the “server-host” is either an IP address or a hostname.

We likely have a root user and an IP address to log in with:

```
ssh root@server-ip
```

On AWS, we might be using user “ubuntu”, and a PEM identity key that AWS has you create and download to your computer:

```
ssh -i ~/.ssh/identity.pem ubuntu@your-server-ip
```

The `-i` flag lets you specify an identity file to use to log in with. Its location on your computer may vary, as AWS has you download it or create it with their API.

In any case, once you’re logged in, you can simply use the `adduser` command to create a new user:

Creating new user ‘someusername’

```
sudo adduser someusername
```

This will ask you for some information, but only the password is required. Take the time to add a lengthy, secure password, but keep in mind you may be asked your password to run privileged commands down the line. You’ll be typing this a lot.

Don’t confuse the `adduser` command with the `useradd` command. On Debian/Ubuntu servers, using `adduser` takes care of some work that we’d have to do manually otherwise.

On Fedora/CentOS/RedHat, `adduser` and `useradd` are the same command (one is symlinked to the other). % signifies a group

Making Our User a Super User

Next, we need to make this new user (“someusername”) a sudo user. This means allowing the user to use “sudo” to run commands as root. How easily you can do this depends on your Linux distribution.

On Ubuntu, you can simply add the user to the pre-existing “sudo” group. We’ll cover users and groups more in a later chapter. Just know for now that all users belong to one or more groups, and groups can be used to manage shared permissions.

Add user 'someusername' to group 'sudo'

```
sudo usermod -a -G sudo someusername
```

Let's go over that command:

- **usermod** - Command to modify an existing user
- **-a** - Append the group to the username's list of secondary groups
- **-G sudo** - Assign the group "sudo" as a secondary group (vs a primary groups, assigned with `-g`)
- **someusername** - The user to assign the group

That's it! Now if we log in as this user, we can use "sudo" with our commands to run them as root. We'll be asked for our users password by default, but then the OS will remember that for a short time. Note that when prompted, you should enter in the *current* user's password, not the password for user "root".

> On RedHat systems, we can add a user to group "wheel" to enable the use of `sudo`:
>
> `sudo usermod -a -G wheel someusername`

On some systems, we likely need to do some extra work to give a new user "sudo" abilities. This is configurable the same way on most systems and is worth covering.

On all distributions mentioned here, there exists the `/etc/sudoers` file. This file controls which users can use `sudo`, and how.

Ubuntu's sudoers file specifies that users within group "sudo" can use the `sudo` command. This provides us with the handy shortcut to granting sudo abilities. On other systems, we'll do this by editing the sudoers file.

We shouldn't edit the sudoers file directly, however. To safely edit this file, use the `visudo` command. Warning: this uses Vim as its editor when opening the file.

If you want to use a friendlier editor, such as **nano**, then we need to set the "EDITOR" environmental variable to "nano". To do so, run `export EDITOR=nano` and then proceed.

Let's begin!

```
sudo visudo
```

Search for a section labeled `# User privilege specification`. Underneath it, you'll likely see something like this:

Editing /etc/sudoers via visudo

```
# User privilege specification
root    ALL=(ALL) ALL
```

This specifies that user "root" can run all commands using sudo with no restrictions.

We can grant another user sudo privileges here:

Editing /etc/sudoers via visudo

```
# User privilege specification
root    ALL=(ALL) ALL
someusername    ALL=(ALL) ALL
```

Similar to user "root", the user "someusername" would now be able to use all sudo privileges. However, this is not *exactly* the same because "someusername" will still need to provide a password to do so.

If you want to setup your server to use the *group* "sudo" to grant sudo privileges, you can set that as well:

Editing /etc/sudoers via visudo

```
# User privilege specification
root    ALL=(ALL) ALL
%sudo   ALL=(ALL) ALL
```

The use of `%` signifies a group name instead of a username. After saving and exiting, we can assign group "sudo" to our new user and they'll also have sudo abilities:

```
# Create group "sudo" if
# it doesn't already exist
sudo groupadd sudo

# Assign someusername the group "sudo"
sudo usermod -a -G sudo someusername
```

More Visudo

Visudo gives us the ability to restrict how users can use the sudo command.

Let's cover using the `/etc/sudoers` file in more detail. Here's an example for user root:

Editing /etc/sudoers via visudo

```
1 root    ALL=(ALL:ALL) ALL
```

Here's how to interpret that. I'll put a [bracket] around each section being discussed. Keep in mind that this specifies under conditions user "root" can use the `sudo` command:

- [root] ALL=(ALL:ALL) ALL - This applies to user root
- root [ALL]=(ALL:ALL) ALL - This rule applies to all user root logged in from all hosts
- root ALL=([ALL]:ALL) ALL - User root can run commands *as* all users
- root ALL=(ALL:[ALL]) ALL - User root can run commands *as* all groups
- root ALL=(ALL:ALL) [ALL] - These rules apply to all commands

As previously covered, you can add your own users:

Editing /etc/sudoers via visudo

```
1 root          ALL=(ALL:ALL) ALL
2 someusername  ALL=(ALL:ALL) ALL
```

We can also set rules for groups. Group rules are prefixed with a `%`:

Editing /etc/sudoers via visudo

```
1 %admin    ALL=(ALL:ALL) ALL
```

Here, users of group `admin` can have all the same sudo privileges as defined above. The group name you use is arbitrary. In Ubuntu, we used group `sudo`.

You may have noticed that in Vagrant, your user can run sudo commands without having to enter a password. That's accomplished by editing the sudoers file as well!

The following entry will allow user `vagrant` to run all commands with sudo without specifying a password:

Editing /etc/sudoers via visudo

```
1 vagrant   ALL=(ALL:ALL) NOPASSWD:ALL
```

The "NOPASSWD" directive does just what it says - all commands run using root do not require a password.

Don't allow users to run ALL commands in production. It makes your privileged user as dangerous as root.

You can get pretty granular with this. Let's give the group "admin" the ability to run 'sudo mkdir' without a password, but require a password to run `sudo rm`:

Editing /etc/sudoers via visudo

```
%admin ALL NOPASSWD:/bin/mkdir, PASSWD:/bin/rm
```

Note that we skipped the `(ALL:ALL)` user:group portion. Defining that is optional and defaults to "ALL".

There's more you can do, but that's a great start on managing how users can use of "sudo"!

Root User Access

Now we have a new user who can use sudo. This is more secure because the user needs to provide their password (generally) to run sudo commands. If an attacker gains access but doesn't know the user's password, then that reduces the damage they can do.

Additionally, this user's actions, even when using sudo, will be logged in their command history. That's not always the case for user "root".

Our next step in securing our server is to make sure we can't remotely (using SSH) log in directly as the root user. To do this, we'll edit our SSH configuration file `/etc/ssh/sshd_config`:

```
# Edit with vim
vim /etc/ssh/sshd_config

# Or, if you're not a vim user:
nano /etc/ssh/sshd_config
```

Use "sudo" with those commands if you're not logged in as "root" currently.

Once inside that file, find the `PermitRootLogin` option, and set it to "no":

File: /etc/ssh/sshd_config

```
PermitRootLogin no
```

Once that's changed, exit and save the file. Then you can restart the SSH process to make the changes take effect:

```
# Debian/Ubuntu:
sudo service ssh restart

# RedHat/CentOS/Fedora:
sudo service sshd restart
```

Now user "root" will no longer be able to login via SSH.

This won't stop user root from logging in directly if the user is physically at the server.

Generally this isn't an issue unless an attacker is at a data center itself! However, some services let you log in directly as root online just as if you're physically next to a computer.

There's still more we can do to secure our servers!

Configure SSH

Many automated bots are out there sniffing for vulnerabilities. One common check is whether the default SSH port is open for connections.

This is such a common attack vector that it's often recommended that you change the default SSH port (22).

This is an example of "Security through obscurity". It is appealing, but found by some to be not worth the effort.

Consider keeping SSH on standard port 22 if it makes sense for and your team. Keep in mind that some software may assume an SSH port of 22. I consider this an optional change.

In "userland", we're allowed to assign ports between 1024 and 65536. To change the SSH port, change the Port option in the same `/etc/ssh/sshd_config` file:

File: /etc/ssh/sshd_config

```
Port 1234
```

Add or edit the `Port` directive and set the port to "1234".

This will tell SSH to no longer accept connections from the standard port 22. One side effect of this is the need to specify the port when you log in later:

```
# Instead of this:
ssh user@hostname

# Add the -p flag to specify the port:
ssh -p 1234 user@hostname
```

We can take this a step further. If we want to explicitly define a list of users who are allowed to login, use the `AllowUsers` directive:

File: /etc/ssh/sshd_config

```
# Can define multiple users,
# separated by a space
AllowUsers someusername anotherusername
```

This tells SSH to only allow logins from the two users listed.

There's also an option to only allow certain groups, using the `AllowedGroups` directive. This is useful for simplifying access - you can add a user to a specific group to decide if they can log in with SSH:

File: /etc/ssh/sshd_config

```
AllowGroups sudo canssh
```

This tells SSH to only allow login from groups "sudo" and "canssh."

Then we can add a user to a secondary group as we saw in an earlier chapter:

```
# Assign secondary group "canssh" to user "ausername"
sudo usermod -a -G canssh ausername
```

Conversely, we can choose to use the `DenyUsers` or `DenyGroups` options. Be careful, however, not to use competing directives.

Once these changes are saved to the `sshd_config` file, we need to restart the SSH service:

```
sudo service ssh restart # Debian/Ubuntu
# OR
sudo service sshd restart # RedHat/CentOS/Fedora
```

Creating a Local SSH Key

We have restricted who can log in, now let's restrict **how** they can log in. User passwords are often "simpler" ones we can remember as we need to use them often. Since passwords are often guessable/crackable, our goal will be to add another layer of security.

What we'll do is disable password-based login altogether, and enforce the use of SSH keys in order to access the server.

Before continuing on, log into the server on a new Terminal window, and keep that connection open. If you get locked out, you'll need this still-open connection to fix any errors.

In order to log in using an SSH key, we need to first create one! What we do is create an SSH key on the computer you need to connect *FROM*.

We'll generate a public and private key, and add the public key to the server. That will let the server know that a private key matching the given public key should allow one to login.

This is more secure. It's substantially less likely for an attacker to get their hands on a local file. Using password-based login, attackers may gain entry by guessing, brute force or social engineering. Your SSH private keys usually only exist on your local computer and thus make it much harder for an attacker to gain entry.

To create an SSH key, run this on your local computer:

```
# Go to or create a .ssh directory for your user
cd ~/.ssh

# Generate an SSH key pair
ssh-keygen -t rsa -b 4096 -C your@email.com -f id_myidentity
```

Let's go over this command:

- **-t rsa** - Create an RSA type key pair[5].
- **-b 4096** - Use 4096 bit encryption. 2048 is "usually sufficient", but I go higher.
- **-C your@email.com** - Keys can have comments. Often a user's identity goes here as a comment, such as their name or email address
- **-f id_myidentity** - The name of the SSH identity files created. The two files would be `id_myidentity` and `id_myidentity.pub` in this example.

[5] http://security.stackexchange.com/questions/23383/ssh-key-type-rsa-dsa-ecdsa-are-there-easy-answers-for-which-to-choose-when

While creating an SSH key, you'll be asked you for a password! You can either leave this blank (for passwordless access) or enter in a password.

I **highly** suggest using a password. Using one forces attackers to have both your private key AND your SSH password to gain access. If your user has sudo abilities, the attacker would also need the user's password to run any sudo command! That's three hard-to-obtain things an attacker would need to get in order to do real damage to your server.

The SSH password you create is NOT the user password used to run sudo commands on the server. It is only used to log into the server. You'll still need the user's regular password to use sudo commands. I recommend not re-using passwords for users and SSH access.

We've created a private key file (`id_myidentity`) and a public key file (`id_myidentity.pub`). Next, we need to put the public key on the server, so that the server knows it's a key-pair authorized to log in.

To do so, copy the contents of the public key file - the one ending in `.pub`. Once that's copied, you can SSH into your server *as your new user* ("someusername" in our example).

Adding public key to authorized_keys file for user 'someusername'

```
# In your server
# Use nano instead of vim, if that's your preference
$ sudo vim ~/.ssh/authorized_keys

# (Paste in your public key and save/exit)
```

I showed editing the file `~/.ssh/authorized_keys`. This will expand out to the full file path for the current user.

Note that we're editing `authorized_keys` for user `someusername`. That means we're enabling ourselves to use SSH-key based login for the `someusername` user.

To gain SSH access, all you need to do is append the public key from our local computer to the `authorized_keys` file of a user on our server.

If there's already content in the `authorized_keys` file, just add your public key in. If the `authorized_keys` file doesn't exist, create it!

> You can use the `ssh-copy-id` command as well, if it's available on your system. MacOS does not have this command out of the box, but can be obtained using the Brew package manager.

Once the `authorized_keys` file is saved, you should be able to login using your key. You shouldn't need to do anything more. Logging in with SSH will attempt your keys first and, finding one, log

in using it, or else fall back to password-based login. You'll need to enter in your password created while generating your SSH key, if you elected to use a password.

If you receive an error when trying to log in, there are two things you can try:

1. Define the identity to use
2. Inform SSH to only use identities (SSH keys), disallowing password-based login attempts

Logging in via SSH key only

```
ssh -i ~/.ssh/my_identity -o "IdentitiesOnly yes" someusername@my-server.com
```

The `-i` flag allows you to define an identity file (SSH private key). The `-o` flag let's you set an "option". In this case we tell SSH to only attempt logins with identify files (SSH keys).

You may also need to set some permissions of your `.ssh` directory and `authorized_keys` file on your server.

The following command should do: `chmod 700 ~/.ssh && chmod 600 ~/.ssh/authorized_keys`

On my Macintosh, I create a long, random SSH password and then save the password to my keychain. Then I don't have to worry about remembering it.

When you log into a server with an SSH key setup for the first time, your Mac should popup asking for your key's password. You'll have the opportunity to save your password to the Keychain then.

If you run into issues SSHing in after this, see the chapter on SSH. Read about using the config file to specify options per SSH connection.

Turn Off Password Access

Since our user can now log in using an SSH key, we no longer need (nor want) to allow users to log in using a user password.

We can tell our server to only allow remote access via SSH keys. To do so, we'll once again edit the `/var/ssh/sshd_config` file within the server:

```
# Use nano instead of vim if you want
sudo vim /etc/ssh/sshd_config
```

Once in the file, find or create the option `PasswordAuthentication` and set it to "no":

```
PasswordAuthentication no
```

Save that change, and once again reload the SSH daemon:

```
sudo service ssh restart # Debian/Ubuntu
# OR
sudo service sshd restart # RedHat/CentOS/Fedora
```

Once that's done, you'll no longer be able to log in using a password! Now a remote attacker will need your SSH private key, your SSH password and your user's password to use `sudo`.

Test it out in a new terminal window to make sure it works! Don't close your current or backup connection, just in case you run into issues and need to revisit your changes.

Note that many providers allow you to access the servers directly in case you lock yourself out of SSH, however many also do not.

In any case, thinking of servers as disposable or prone to fail is always a pertinent thing to do.

Backup important files, configurations and data somewhere else - off of the server. Amazon AWS's S3 service is an excellent, cheap place to put backups.

Setting Up the Firewall: Iptables

The firewall offers some really important protections on your server. Firewalls will block network traffic as defined by a set of rules.

While iptables is the defacto firewall used on most Linux distributions, it is a little hard to pick up and use.

Configuring iptables involves setting up the list of rules that check network traffic. The rules are checked whenever a piece of data enters or leaves the server over a network. If the iptables rules allows the traffic type, it goes through. If traffic is not allowed, the data packet is dropped or rejected.

Rejecting data lets the other end know data was not allowed through. Dropping the data behaves like a blackhole, where no response is made.

The following is a basic list of INPUT (inbound) rules we'll be building in this chapter:

Results of command `iptables -L -v`

```
1  target     prot opt in     out     source      destination
2  ACCEPT     all  --  lo     any     anywhere    anywhere
3  ACCEPT     all  --  any    any     anywhere    anywhere      ctstate RELATED,EST\
4  ABLISHED
5  ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:ssh
6  ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:http
7  ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:https
8  DROP       all  --  any    any     anywhere    anywhere
```

This is some of the output from the command `sudo iptables -L -v`. This command lists the rules with verbosity.

Let's go over the columns we see above:

1. TARGET: What to do with the traffic and/or other chains of rules to test traffic against
2. PROT: Protocol, usually "tcp", "udp" or "all". TCP is the most used. SSH and HTTP are protocols built on top of TCP.
3. OPT: Optional items, such as checking against fragmented packets of data
4. IN: Network interface accepting traffic, such as lo, eth0, eth1. Check what interfaces exist using the `ifconfig` command.

5. OUT: Network interface the traffic goes out
6. SOURCE: The source of some traffic, such an a hostname, ip address or range of addresses
7. DESTINATION: The destination address of the traffic

These rules are followed in order. The first rule that matches the traffic type will determine what happens to the data.

Let's go over the above list of rules we have for inbound traffic, in order of appearance:

1. Accept all traffic on "lo", the "loopback" interface[6]. This is essentially saying "Allow all **internal** traffic to pass through"
2. Accept all traffic from currently established (and related) connections. This is set so you don't accidentally block yourself from the server when in the middle of editing firewall rules
3. Accept TCP traffic over port 22 (which iptables labels "ssh" by default). The port defined as the SSH port is defined in the `/etc/services` file.
4. Accept TCP traffic over port 80 (which iptables labels "http" by default)
5. Accept TCP traffic over port 443 (which iptables labels "https" by default)
6. Drop anything and everything else

See how the last rule says to DROP all from/to anywhere? If a packet has passed all other rules without matching, it will reach this rule, which says to DROP any and all data.

The effect is that we're only allowing current connection, SSH (tcp port 22), http (tcp port 80) and https (tcp port 443) traffic into our server! The DROP statement blocks everything else.

The first rule that matches the traffic type will decide how to handle the traffic. Rules below a match are not applied.

"Traffic Type" includes protocol, interface, source/destination and other parameters.

If more than one rule match the traffic type, the 2nd rule is never reached.

We've effectively protected our server from external connections other than TCP port 22, 80 and 443.

Adding these rules

Now you need to know how to add these rules. First, check your current set of rules by running the following:

[6] http://askubuntu.com/questions/247625/what-is-the-loopback-device-and-how-do-i-use-it

```
sudo iptables -L -v
```

If you have no firewalls rules setup, you'll see something like this:

```
Chain INPUT (policy ACCEPT 35600 packets, 3504K bytes)
 pkts bytes target     prot opt in     out     source    destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source    destination

Chain OUTPUT (policy ACCEPT 35477 packets, 3468K bytes)
 pkts bytes target     prot opt in     out     source    destination
```

What we see above are the three default *chains* of the filter table:

1. `INPUT` chain - Traffic inbound to the server
2. `FORWARD` chain - Traffic forwarded (routed) to other locations
3. `OUTPUT` chain - Traffic outbound from the server

The ArchWiki has a great explanation on Tables vs Chains vs Rules[7]. There are other tables/chains as well - see NAT, Mangle and Raw tables[8]

Let's add to our Chain of rules by appending to the INPUT chain. First, we'll add the rule to allow all loopback traffic:

```
sudo iptables -A INPUT -i lo -j ACCEPT
```

The details of the above command:

- `-A INPUT` - **A**ppend to the **INPUT** chain
- `-i lo` - Apply the rule to the **lo**opback **i**nterface
- `-j ACCEPT` - **J**ump the packet to the ACCEPT rule. Basically, accept the data packets. "ACCEPT" is a built-in "target", but you can jump to user-defined ones as well (more on that later)

Now let's add the rule to accept current/established connections:

[7] https://wiki.archlinux.org/index.php/iptables
[8] http://www.thegeekstuff.com/2011/01/iptables-fundamentals/

```
sudo iptables -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

And the command explanation:

- `-A INPUT` - **A**ppend to the INPUT chain
- `-m conntrack` - **M**atch traffic using "connection tracking" module
- `--ctstate RELATED,ESTABLISHED` - Match traffic with the **state** "established" and "related"
- `-j ACCEPT` - Use the ACCEPT target; accept the traffic

This one's a little on the complex side but I won't focus on it here. If you're curious about "conntrack" and other modules, you can search for "iptables modules".

Let's start adding the more interesting rules. We'll start by opening up our SSH port for remote access:

```
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

And the command explanation:

- `-A INPUT` - **A**ppend to the INPUT chain
- `-p tcp` - Apply to the tcp **P**rotocol
- `--dport 22` - Apply to **d**estination **port** 22 (Incoming traffic coming into port 22).
- `-j ACCEPT` - Use (**j**ump to) the ACCEPT target; accept the traffic

If you check your rules after this with another call to `sudo iptables -L -v`, you'll see that "22" is labeled "ssh" instead. If you don't use port 22 for SSH, then you'll see the port number listed.

We can add a very simlar rule for HTTP traffic:

```
sudo iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

And lastly, we'll add the "catch all" to DROP any packets which made it this far down the rule chain:

```
sudo iptables -A INPUT -j DROP
```

And the command explanation:

- `-A INPUT` - **A**ppend to the INPUT chain
- `-j DROP` - Use the DROP target; deny the traffic

You could use `-j REJECT` as well. `REJECT` explicitly tells the client the data came from that the data wasn't accepted.

Using `DROP` yields no response to the client. When `DROP` is used, a client usually reaches a connection timeout since it receives no response from the server.

Inserting Rules

So far we've seen how to **A**ppend rules (to the bottom of the chain). Let's see how to **I**nsert rules, so we can add rules in the middle of a chain.

We haven't yet added a firewall rule to allow HTTPS traffic (port 443). Let's do that:

```
sudo iptables -I INPUT 5 -p tcp --dport 443 -j ACCEPT
```

And the command explanation:

- `-I INPUT 5` - **I**nsert into the INPUT chain at the fifth position. This is just after the "http" rule at the fourth position. Position count starts at 1 rather than 0.
- `-p tcp` - Apply the rule to the tcp **p**rotocol
- `--dport 443` - Apply to the **d**estination port 443 (Incoming traffic coming into port 443).
- `-j ACCEPT` - Use the ACCEPT target; accept the traffic

Deleting Rules

Let's say we want to change our SSH port from the non-standard port 22. We'd set that in `/etc/ssh/sshd_config` as explained in the Users and Access chapter. Then we would need to change the firewall rules to allow SSH traffic to our new port (port 1234 in this example).

First, we'll delete the SSH rule:

```
# Delete at position 3
sudo iptables -D INPUT 3

# Or delete by specifying the rule to match:
sudo iptables -D INPUT -p tcp --dport 22 -j ACCEPT
```

We can see that `-D` will delete the firewall rule. We need to either match the position of the rule or all the conditions set when creating the rule to delete it.

Then we can insert our new SSH rule at port 1234:

```
sudo iptables -I INPUT 3 -p tcp --dport 1234 -j ACCEPT
```

This chapter covers ipv4 IP addresses. Iptables can handle rules for both ipv4[9] and ipv6[10], however.

Now check that we've accomplished all that we've wanted:

[9] http://en.wikipedia.org/wiki/IPv4
[10] http://en.wikipedia.org/wiki/IPv6

```
$ sudo iptables -L -v
```

The output:

```
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target  prot opt in     out     source      destination
3226  315K ACCEPT  all  --  lo     any     anywhere    anywhere
712 37380 ACCEPT   all  --  any    any     anywhere    anywhere      ctstate REL\
ATED,ESTABLISHED
0     0 ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:ssh
0     0 ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:http
0     0 ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:htt\
ps
8  2176 DROP       all  --  any    any     anywhere    anywhere
```

Perfect! It's just like our desired list of rules from the beginning of the chapter.

Output Rules as Commands

We can use `sudo iptables -S` to get a list of the current rules given as commands. You can then copy and paste a given rule output in order to match a rule for deletion or to use the rules elsewhere.

Let's see an example:

```
$ sudo iptables -S

-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m tcp --dport 22 -j ACCEPT
-A INPUT -p tcp -m tcp --dport 80 -j ACCEPT
-A INPUT -p tcp -m tcp --dport 443 -j ACCEPT
-A INPUT -j DROP
```

This outputs our current rule set as the commands we can use to create the rules.

If we wanted to remove the SSH rule again, we could copy and paste it, change the `-A` (Append) to `-D` (Delete) and be done with it:

```
$ sudo iptables -D INPUT -p tcp -m tcp --dport 22 -j ACCEPT
```

Saving Firewall Rules

By default, iptables does **not** save firewall rules after a reboot, as the rules exist only in memory. We therefore need a way to save the rules and re-apply them on reboot.

At any time, you can print out the current iptables rules:

```
sudo iptables-save
```

We can restore iptables rules using the `iptables-restore` command:

```
# Output rules to a file called "iptables-backup.rules"
sudo iptables-save > iptables-backup.rules

# Restore rules from our backup file
sudo iptables-restore < iptables-backup.rules
```

What we need is a way to automate the backing up and restoration of firewall rules, preferably on system boot.

On Ubuntu, we can use the `iptables-persistent` package to do this:

```
# Install the package
sudo apt-get install -y iptables-persistent

# Start the service
sudo service iptables-persistent start
```

Once this is installed, we can output our rules to the `/etc/iptables/rules.v4` file. Iptables-persistent will read this file when the system starts.

We'll use the `iptables-save` command to output the rules. This output will be saved to the `/etc/iptables/rules.v4` file.

Save current rules to iptables rules file

```
sudo iptables-save | sudo tee /etc/iptables/rules.v4
```

If you are using ipv6, you can use `sudo ip6tables-save | sudo tee /etc/iptables/rules.v6` with the iptables-persistent package.

When that's done, restart iptables-persistent:

```
sudo service iptables-persistent restart
```

Now firewall rules will be re-applied on a server restart! Don't forget to update your rules files after any firewall changes.

The `iptables-persistent` package has a shortcut for the above. We can simply use `sudo service iptables-persistent save` to save our current ruleset.

Defaulting to DROP Over ACCEPT

So far, we've seen one method of using iptables. The default for each chain is to ACCEPT traffic. Notice that when we list the rules, we can see that - "policy ACCEPT":

```
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
```

Debian/Ubuntu servers usually start with the chains defaulting to ACCEPT. However, Redhat/CentOS servers may start with their chains defaulting to DROP traffic. Defaulting to DROP can often be easier (and safer). Let's see how we can do that.

Let's change the INPUT chain to default to DROP:

```
sudo iptables -P INPUT DROP
```

Then we can remove the last line used above, which DROPs any remaining unmatched rules:

```
sudo iptables -D INPUT -j DROP
```

If we run `iptables -L`, we can see the INPUT chain now defaults to DROP:

```
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ES\
TABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:ssh
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:http
```

Now everything is going to be dropped unless explicitly accepted.

Overall, some general rules of thumb for the three chains are to:

- Drop traffic on the INPUT chain by default
- Drop traffic on the FORWARD chain by default
- Allow traffic on the OUTPUT chain by default

Logging Dropped Packets

You might find it useful to log dropped packets (traffic). To do this, we'll actually create a new chain. Here are the basic steps:

1. Create a new chain
2. Ensure any unmatched traffic 'jumps' to the new chain
3. Log the packets with a searchable prefix
4. Drop those packets

Let's start!

```
# Create new chain
sudo iptables -N LOGGING

# Ensure unmatched packets jump to new chain
sudo iptables -A INPUT -j LOGGING
```

At this point, you should delete any DROP rule that might be at the end of the INPUT chain. That might still be there if you followed along in the above sections.

```
sudo iptables -D INPUT -j DROP
```

Then continue on:

```
# Log the packets with a prefix
sudo iptables -A LOGGING -m limit --limit 2/min -j LOG --log-prefix "IPTables Pa\
cket Dropped: " --log-level 7

# Drop those packets
# Note this is added to the LOGGING chain
sudo iptables -A LOGGING -j DROP
```

Here's what that'll look like when we run `iptables -L -v`:

```
Chain INPUT (policy DROP)
target     prot opt source      destination
ACCEPT     all  --  anywhere    anywhere      ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere    anywhere
ACCEPT     tcp  --  anywhere    anywhere      tcp dpt:ssh
ACCEPT     tcp  --  anywhere    anywhere      tcp dpt:http
LOGGING    all  --  anywhere    anywhere

Chain LOGGING (1 references)
target     prot opt source      destination
LOG        all  --  anywhere    anywhere      limit: avg 2/min burst 5 LOG level\
 debug prefix "IPTables Packet Dropped: "
DROP       all  --  anywhere    anywhere
```

Note that we DROP the data in the LOGGING chain. The INPUT chain is no longer responsible for dropping data. Instead, any traffic that doesn't match the rules in the INPUT chain "jumps" to the LOGGING chain to be logged and *then* dropped.

By default, this will go to the kernel log. In Ubuntu, that means we can watch the log file `/var/log/kern.log`:

```
sudo tail -f /var/log/kern.log
```

I see entries like this when attempting connections which get dropped. This is an example log for HTTPS traffic which gets dropped on one of my servers, which blocks port 443:

```
Dec  5 02:27:51 precise64 kernel: [ 2101.687289] IPTables Packet Dropped: IN=eth\
1 OUT= MAC=08:00:27:4f:82:c9:0a:00:27:00:00:00:08:00 SRC=192.168.33.1 DST=192.16\
8.33.10 LEN=64 TOS=0x00 PREC=0x00 TTL=64 ID=59982 DF PROTO=TCP SPT=51765 DPT=443\
 WINDOW=65535 RES=0x00 SYN URGP=0
```

If you set this up, don't forget to save these rules as noted above, using the `iptables-persistent` package:

```
sudo iptables-save | sudo tee /etc/iptables/rules.v4
```

Fail2Ban

There is one more important security tool we'll cover: Fail2Ban.

Fail2Ban monitors for instrusion attempts on your server. It uses the iptables firewall to ban specific hosts if they meet a configured threshold of invalid attempts.

Fail2Ban does this by monitoring the log files of certain services. For example, Fail2Ban will monitor logs found at `/var/log/auth.log` and search for failed logins. If it detects a host has failed to login too many times, it will ban that host for a configurable time period.

Here's the explanation from the website[11]:

> Fail2Ban scans log files (e.g. /var/log/apache/error_log) and bans IPs that show the malicious signs – too many password failures, seeking for exploits, etc. Generally Fail2Ban is then used to update firewall rules to reject the IP addresses for a specified amount of time, although any arbitrary other action (e.g. sending an email) could also be configured. Out of the box Fail2Ban comes with filters for various services (apache, courier, ssh, etc).

Iptables Integration

When Fail2Ban bans a host, it will use the iptables firewall.

Some terminology: Each system Fail2Ban monitors is called a "jail". For example, one jail is called "SSH", another is "mysqld-auth".

To do this, Fail2Ban creates a new iptables chain per jail it monitors. For SSH, Fail2Ban will create a chain called "Fail2Ban-ssh". This chain (and others it creates) is used early in the iptables INPUT chain, so it gets checked first. Let's see what that looks like after Fail2Ban is configured:

[11] http://www.fail2ban.org

```
$ sudo iptables -L -v

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target       prot opt in   out  source    destination
 123K 123M  Fail2Ban-ssh tcp  --  any  any  anywhere  anywhere  multiport dports\
 ssh
 292K  169M ACCEPT       tcp  --  any  any  anywhere  anywhere  tcp dpt:http
... additional omitted ...


Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 ... omitted ...


Chain OUTPUT (policy ACCEPT 939K packets, 2332M bytes)
 ... omitted ...


Chain Fail2Ban-ssh (1 references)
 pkts bytes target     prot opt in     out     source     destination
1962K 1498M RETURN     all  --  any    any     anywhere   anywhere
```

Here's what's happening. When traffic comes **into** the network, iptables checks it by going down the list of rules. Since we're talking about incoming traffic, this means iptables will check the INPUT chain.

Fail2Ban adds the first rule in the above INPUT chain. It says to take all SSH traffic and send it to the target chain `Fail2Ban-ssh`. The Fail2Ban-ssh chain then checks for any matching hosts and DROPs the traffic if any match.

In the example above, there happens to be no hosts being blocked, so any traffic being checked will meet the "RETURN" target. The "RETURN" target simply tells iptables to send the traffic back to where it came from - the INPUT chain in this case. There it will be analyzed by the rest of rules in the INPUT chain.

Now that we can see how Fail2Ban will use iptables, let's see how to install and configure Fail2Ban!

Installation

We don't need a repository for Fail2Ban, we can just install it straight away.

```
sudo apt-get install -y fail2ban
```

Fail2Ban's configuration files are found in `/etc/fail2ban`. Fail2Ban comes with the default configuration file `/etc/fail2ban/jail.conf`. This file might get updated/overwritten on updates, so we should copy it instead of editing it directly.

Fail2Ban will automatically check for a file named `jail.local` and load it, so let's use that filename:

```
1 sudo cp /etc/fail2ban/jail.conf /etc/fail2ban/jail.local
```

This new file will serve as our main configuration.

You might also notice a `/etc/fail2ban/jail.d` directory. This is a directory into which we can add configurations that will also be enabled automatically.

Configuration files added in `/etc/fail2ban/jail.d` can tweak or overwrite configuration found in `/etc/fail2ban/jail.local`.

From the manual on Fail2Ban:

> In addition to .local, for any .conf file there can be a corresponding .d/ directory to contain additional .conf files that will be read after the appropriate .local file. Last parsed file will take precedence over identical entries, parsed alphabetically…

Files in `jail.d` can overwrite existing configurations and add additional configurations. However, we'll simply use the `jail.local` file for our purposes.

Configuration

Once we've copied the `jail.conf` file to `jail.local`, we can take a look inside. We'll review interesting (not all!) configurations. Open up the `/etc/Fail2Ban/jail.local` file:

Here are some items under the `[DEFAULT]` section:

ignoreip

The `ignoreip` directive is usually set to `127.0.0.1/8`, which will ignore local network connections. You can define multiple hosts as well. Use a space to separate hosts. CIDR notation is supported.

bantime

The `bantime` directive defaults to 600 seconds, and is the amount of time a host will be banned.

findtime

This also defaults to 600 seconds. The `findtime` directive is the amount of time wherein a threshold must be met in order to trigger a ban.

maxretry

The `maxretry` directive defaults to 3. It specifies the number of bad attempts before a ban is triggered.

This works in conjunction with `findtime`. Using the defaults, 3 incorrect attempts within 600 seconds will trigger a ban.

Action Shortcuts

Fail2Ban sets up some pre-built actions to take when a ban is triggered. The default ("actions_") is used to simply ban the host. Other actions allow you to email a whois report ("actions_mw"), or a whois report + relevant log lines ("actions_mwl").

The `action` directive will set which action to use. The default `actions_` looks like this:

Set the action to simply ban the host

```
1 action = %(action_)s`
```

Sending an email of the WHOIS record would look like this:

Set the action to ban the host and send an email

```
1 action = %(actions_mw)s
```

In order to have Fail2Ban email you when a ban occurs, the system can use the conventional `mail` command or can use `sendmail`. The `mta` directive found in `jail.local` sets this. Sendmail will need to be installed and setup separately for this to work.

Also set the `destemail` directive to the email address you want ban reports sent to.

I typically do NOT setup email reports. A bannings is common - many (most) servers on a public network will have multiple unsuccessful login attempts daily.

Jails

The last part of the configuration file is setting up the jails. Each Jail specifies a service to be monitored. By default, we will see only SSH is enabled:

```
[ssh]

enabled  = true
port     = ssh
filter   = sshd
logpath  = /var/log/auth.log
maxretry = 6
```

To quickly cover the directives above:

- It's named "ssh"
- It's enabled
- It monitors services on the SSH port
- It monitors the logs at the logpath `/var/log/auth.log`
- It over-rides the default max retries, increasing them to 6.

The `filter` directive refers to the filter used to scrum through the log file. The filter named `sshd` correlates directly to the `/etc/fail2ban/filter.d/sshd.conf` filter file. The filter is used to determine which lines in the log constitutes a login failure.

If you inspect that filter file, you'll see some regex being used to determine which line in the inspected log is a failed retry.

There are many Jails defined, but only the SSH jail is enabled by default - it's one of the most important and commonly used jails. There are, of course, others that you may want to enable, such as:

- `ssh-ddos` - Protection from SSH-based denial of service attacks, usually coming in the form of connection attempts with no identities
- `apache` - If you're using Apache, there is a suite of protections you can enable. These protect against Apache basic-auth attempts (`[apache]`), attempts to run script files (such as php) (`[apache-noscript]`), and memory buffer overflow attacks (`[apache-overflow]`).
- `dovecot` - Among other email protection filters, this one helps detect intrusion attempts against the Dovecot SMTP server
- `mysqld-auth` - This helps protect against too many incorrect logins for MySQL.

Reading the filter file for a jail is a great way to ascertain what the jail is attempting to protect against.

Nginx

There's only one Nginx jail defined out of the box. It's used to protect Nginx against HTTP basic-auth attacks:

```
[nginx-http-auth]

enabled = true
filter  = nginx-http-auth
port    = http,https
logpath = /var/log/nginx/*error.log
```

This uses the "nginx-http-auth" filter which comes with Fail2Ban. For other Nginx jails and filters to add yourself, check here[12].

Once a jail is enabled or when a configuration is edited, you can reload Fail2Ban so the changes will take affect:

```
sudo service fail2ban reload
```

Logs for Fail2Ban actions can be found at `/var/log/fail2ban`. Keep an eye on these logs to monitor what intrusion attempts are made on your server(s)!

[12] http://snippets.aktagon.com/snippets/554-how-to-secure-an-nginx-server-with-fail2ban

Automatic Security Updates

You may want your server to automatically update software. Most distributions of Linux allow you to set this up.

Automated updates can be dangerous, however. We do not always want to update *all* software without first testing the updates, as we never know what might cause issues.

> This tip comes from hard experience. Before Ubuntu 14.04, the `ppa:ondrej/php5` repository allowed us to install PHP 5.5 on Ubuntu 12.04. I was quick to upgrade when this repository made PHP 5.5 available.
>
> However, this also updated the version of Apache required. I inadvertently updated Apache from version 2.2 to 2.4! The newer version of Apache had breaking configuration changes, and so brought down my sites.

In Ubuntu, we can choose to enable only automatic *security* updates. This reduces the risk of non-essential updates causing issues.

Whether you consider this a best practice is up to you. Perhaps security updates have potential to break your applications. Use this as you see fit. I personally have it enabled on my own servers.

If you want to enabled security upgrades, first ensure the `unattended-upgrades`[13] package is installed:

```
1 sudo apt-get install -y unattended-upgrades
```

Then update `/etc/apt/apt.conf.d/50unattended-upgrades`. The number preceding the filename might vary a bit. Make sure `"Ubuntu trusty-security";` is enabled. The remaining "Allowed-Origins" listed can be deleted or commented out:

File: /etc/apt/apt.conf.d/50unattended-upgrades

```
1 Unattended-Upgrade::Allowed-Origins {
2     "Ubuntu trusty-security";
3 //  "Ubuntu trusty-updates";
4 };
```

My example says "trusty" since I'm using Ubuntu 14.04. You might have a different name for your Ubuntu distribution there, such as "precise" (12.04).

Alternatively, you might see the following inside of the `/etc/apt/apt.conf.d/50unattended-upgrades` file:

[13] https://help.ubuntu.com/14.04/serverguide/automatic-updates.html

File: /etc/apt/apt.conf.d/50unattended-upgrades, Allowed-Origins

```
Unattended-Upgrade::Allowed-Origins {
    "${distro_id}:${distro_codename}-security";
//  "${distro_id}:${distro_codename}-updates";
//  "${distro_id}:${distro_codename}-proposed";
//  "${distro_id}:${distro_codename}-backports";
};
```

If you see this you're all set. The above configuration handles changing for your distribution of Ubuntu dynamically.

Some updates can trigger a server reboot; You should decide if you want upgrades to be able to do so:

File: /etc/apt/apt.conf.d/50unattended-upgrades

```
`Unattended-Upgrade::Automatic-Reboot "false";
```

Be careful with allowing servers to restart automatically. Your applications or processes may not be configured to restart when a server reboots. See the chapter "Monitoring Processes" for more information.

Finally, create or edit the `/etc/apt/apt.conf.d/02periodic` file and ensure these lines are present:

File: /etc/apt/apt.conf.d/02periodic

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Download-Upgradeable-Packages "1";
APT::Periodic::AutocleanInterval "7";
APT::Periodic::Unattended-Upgrade "1";
```

Once that's complete, you're all set!

This will run once at set intervals. "Periodic" items are set to run once per day via the daily cron. If you're curious, you can find that configured in the `/etc/cron.daily/apt` file.

Upgrade information is logged within the `/var/log/unattended-upgrades` directory.

Package Managers

We've installed quite a bit of software already. Before we continue on any further, let's talk about package managers more in depth.

Package Managers install software onto our servers. To do so successfully, they must serve three important functions:

1. Install the software version appropriate for the distribution (operation system) and OS version
2. Manage dependencies required by software, including finding and attempting to fix dependency issues
3. Add configurations to gracefully start and stop as the server restarts. This includes process monitoring to keep them alive in case of errors.

On Debian/Ubuntu, we'll be dealing with the APT package manager.

Apt

As previously stated, this book concentrates on Debian/Ubuntu. Therefore, we'll installing software with APT.

APT stands for Advanced Packaging Tool.

Installing

Apt keeps a list of sources on the server. Each source contains lists of repositories. The repositories serve as indexes of available packages. Apt will check against this list when you search for packages to install.

The sources, and their lists of repositories, are kept in two places:

- The `/etc/apt/sources.list` file
- Files inside of the `/etc/apt/sources.list.d` directory

We can update Apt's knowledge of available package and versions by running the following command:

```
sudo apt-get update
```

This will read the list of repositories and update the packages and versions available to install.

Run this before installing any software or after adding new repositories. This will ensure it installs the most recent available version of a package.

Once the source lists are updated, we can install whatever software we'd like, based on their package name.

```
sudo apt-get install some-package
```

Here are some useful flags to use with the install command:

- `-y`/`--yes` - Skip prompts asking if you're sure you want to install the package
- `--force-yes` - Install even when there are potential issues. One such issue is the package not being "trusted"
- `-qq` - Quiets some output, except for errors and basic installation information. Implicitly means both `-y` and `--force-yes`

Repositories

In Ubuntu, there will be software and security updates within the two years between LTS releases.

These are often incorporated into minor updates (for example 14.04.1, 14.04.2, and so on). However, if there are security or feature updates that we can't wait on, how would we go about getting them?

One complex way is to download and build the software manually. This, however, circumvents all the things we like about using a package manager. Configuration, process monitoring, starting on boot, and dependency checking are skipped!

An easier way of getting updates is to add package repositories to our source list. This lets us get software updates that wouldn't normally be available on our server version.

We can add third-party repositories manually or use the `add-apt-repository` command.

The `add-apt-repository` command will add a source to `/etc/apt/sources.list` or `/etc/apt/-sources.list.d`. The repository added will be appropriate for our server version.

```
# Installs 'add-apt-repository', although it's likely already installed
sudo apt-get install software-properties-common

# Add a repository
sudo add-apt-repository -y ppa:namespace-name/repo-name
```

Just like for `apt-get`, the `-y` flag is to answer "yes" to any "are you sure?" type prompts.

Examples

Let's use installing Redis as an example.

On first glance, we can see that there is a Redis package available named "redis-server":

Searching for a redis package

```
# Search for a redis package:
sudo apt-cache -n search redis
...
redis-server
redis-tools
...
```

Let's get some information about the redis-server package:

Showing information on the 'redis-server' package

```
$ apt-cache show redis-server
Package: redis-server
Priority: optional
Section: universe/misc
Installed-Size: 744
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Chris Lamb <lamby@debian.org>
Architecture: amd64
Source: redis
Version: 2:2.8.4-2
Depends: libc6 (>= 2.14), libjemalloc1 (>= 2.1.1), redis-tools (= 2:2.8.4-2), ad\
duser
Filename: pool/universe/r/redis/redis-server_2.8.4-2_amd64.deb
Size: 267446
MD5sum: 066f3ce93331b876b691df69d11b7e36
SHA1: f7ffbf228cc10aa6ff23ecc16f8c744928d7782e
SHA256: 2d273574f134dc0d8d10d41b5eab54114dfcf8b716bad4e6d04ad8452fe1627d
Description: Persistent key-value database with network interface
Description-md5: 9160ed1405585ab844f8750a9305d33f
Homepage: http://redis.io/
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Origin: Ubuntu
```

Here's a bunch of information on the redis-server package. We can see the available version as well as information on dependencies and the maintainer.

To check to see what versions are available to install, using the command `apt-cache policy`:

Checking the current policy for the 'redis-server' package

```
sudo apt-cache policy redis-server
redis-server:
  Installed: (none)
  Candidate: 2:2.8.4-2
  Version table:
     2:2.8.4-2 0
        500 http://archive.ubuntu.com/ubuntu/ trusty/universe amd64 Packages
```

There's only one version in the version table: 2.8.4.

Looking at the Redis official site, however, we can see that version 2.8.12 is available (as of the time of this writing). How might we get a newer version?

Searching around the web, we can find the redis repository from Chris Lea[14]. This repository has version 2.8.12 available!

Let's add this repository in to get the newer version of Redis:

```
sudo add-apt-repository ppa:chris-lea/redis-server
```

After adding this, you'll find a new source list file: `/etc/apt/sources.list.d/chris-lea-redis-server-trusty.list`

Then we can update our local repository list:

```
sudo apt-get update
```

And re-check the available versions:

```
apt-cache policy redis-server
redis-server:
  Installed: (none)
  Candidate: 2:2.8.12-1chl1~trusty1
  Version table:
     2:2.8.12-1chl1~trusty1 0
        500 http://ppa.launchpad.net/chris-lea/redis-server/ubuntu/ trusty/main \
amd64 Packages
     2:2.8.4-2 0
        500 http://archive.ubuntu.com/ubuntu/ trusty/universe amd64 Packages
```

Great, 2.8.12 is now our candidate! Let's install it:

```
sudo apt-get install -y redis-server
```

Now we have a more up-to-date repository for Redis, which will usually be ahead of the server's released (out of the box) version. This can help if and when there are vital security, bug fix or feature updates we need.

Another popular redis repository is ppa:rwky/redis[15] which may sometimes contain a slightly newer version.

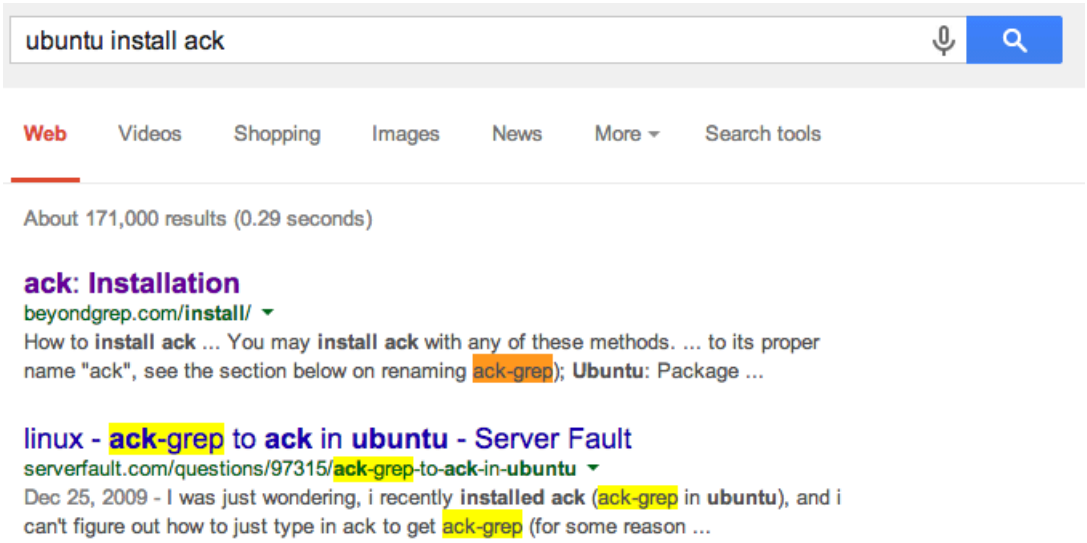
[14] https://launchpad.net/~chris-lea/+archive/redis-server
[15] https://launchpad.net/~rwky/+archive/ubuntu/redis

Searching Packages

In the Basic Software chapter of the Getting Started section, we installed "Ack". However, I used "ack-grep" as the name of the package to install. How did I know its package name? I had to search for it!

There are two methods of searching packages:

Google. This is usually the fastest way! For example, the query "ubuntu install ack" gets you the answer straight-away:



Alternatively, we can use `apt-cache` to search for packages.

Search for a package to install 'ack'

```
1 apt-cache search ack
```

That lists out a HUGE amount of possibilities. It's actually searching package names and descriptions. Let's try to narrow that down by searching package names only.

```
1 # -n flag searches only package names
2 apt-cache search -n ack
```

This output (still not shown here) is better, but still too large. Let's try to get the results in alphabetical order. We can "pipe" the output to the `sort` command and then search for "ack" using `grep`. Grep is a tool for searching through text. It usually gives some color to text matches in the terminal output, making "ack" easier to spot:

```
apt-cache search -n ack | sort | grep ack
```

The output:

```
ack-grep - grep-like program specifically for large source trees
aircrack-ng - wireless WEP/WPA cracking utilities
akonadi-backend-mysql - MySQL storage backend for Akonadi
akonadi-backend-postgresql - PostgreSQL storage backend for Akonadi
akonadi-backend-sqlite - SQLite storage backend for Akonadi
alsaplayer-jack - PCM player designed for ALSA (JACK output module)
apt-watch-backend - Applet that monitors apt sources for upgrades (backend slave\
)'
...
```

There's still a lot of output, but at least "ack-grep" is now on top! The package name is `ack-grep`, and that's what we can use to install it using the `apt-get install` command.

Permissions and User Management

Permissions

Permissions in Linux can be a bit confusing at first. Every directory and file have their own permissions. Permissions inform the system who and how users can perform operations on a file or directory.

Users can perform **read (r)**, **write (w)** and **execute (x)** operations on files and directories. Here's how the three permission types breaks down when applied to directories and files:

- **Directories**
  - `read` - ability to read contents of a directory
  - `write` - ability to rename or create a new file/directory within a directory (or delete a directory)
  - `execute` - ability to `cd` into a directory (this is separate from being able to view the contents of a directory)
- **Files**
  - `read` - ability to read a file
  - `write` - ability to edit/write to a file (or delete a file)
  - `execute` - ability to execute a file (such as a bash command)

The other half of this is defining who (what users and groups) can perform these operations. For any file and directory, we can define how **users (u)**, **groups (g)** and **others (o)** can interact with the file or directory. Here's how that breaks down:

- `User` - The permission for *owners* of a file or directory
- `Group` - The permissions for users belonging to a *group*. A user can be part of one or more groups. Groups permissions are the primary means for how multiple users can read, write or execute the same sets of files
- `Other` - The permissions for users who aren't the user or part of a group assigned to a file or directory

Checking Permissions

To illustrate this, let's check the permissions of a directory, for example `/var/www`:

```
$ ls -la /var/www
drwxr-xr-x  2 root root 4096 May  3 19:52 .          # Current Directory
drwxr-xr-x 12 root root 4096 May  3 19:46 ..         # Containing Directory
-rw-r-xr--  1 root root   13 May  3 19:52 index.html # File in this Directory
```

How do these columns of information break down? Let's take the top line:

- `drwxr-xr-x` - User/Group/Other Permissions. The preceding "d" denotes this as a directory. Lacking a "d" means it's a file.
- `2` - This is the number of "hard links"[16] to the file or directory
- `root root` - The User and Group assigned to the file or directory
- `4096` - The size of the file/directory in bytes
- `May 3 19:52` - last modified (or created) data/time
- `.` - The file name. A period (.) is the current directory. Two periods (..) is the directory one level up. Otherwise this column will show a file or directory name.

Let's go over the permission attributes - that first column of information:

For any permission attribute set, the first slot denotes if it's a **directory (d)**, **link (l)** (as in symbolic link) or **file (-)**.

The next three sets of characters denote the read, write and execute permissions for users groups and others, respectively.

Let's take the permissions `drwxr-xr-x`.

- `d` - denotes it's a directory
- `rwx` - The user has read, write and execution permissions
- `r-x` - The group can read and execute (list contents and cd into the directory), but not write to the directory
- `r-x` - The same for others. Since this is a directory, this means other users can read the directory but not modify it or its containing files

Next, let's analyze `-rw-r-xr--`:

- `-` - denotes it's a file
- `rw-` - denotes users can read, write but not execute the file
- `r-x` - group members can read the file or execute it
- `r--` - others can only read the file

Changing Permissions

We can change a file or directory permissions with the `chmod` command.

Here's some chmod information and a breakdown:

[16] http://superuser.com/a/443781

```
chmod [-R] guo[+-=]rwx /var/www
```

Flags:

- `-R` - Change permissions recursively (if its a directory)

User types:

- `u` - perform operation on the user permissions
- `g` - perform operation on the group permissions
- `o` - perform operation on the other permissions

Operations:

- `+` - add permission
- `-` - remove permission
- `=` - set permission explicitly

Permission types to set

- `r` - add or remove read permissions
- `w` - add or remove write permissions
- `x` - add or remove execute permissions

So, for example, let's create the `/var/www` directory as user root and set its permissions.

```
# Create directory as root
sudo mkdir /var/www

# Change to owner/group www-data:
sudo chown www-data:www-data /var/www

# Set permissions so user and group has permissions, but not other:
sudo chmod ug+rwx /var/www # User and Group have all permissions (+rwx)
sudo chmod o-rwx /var/www  # Other has no permissions (-rwx)
sudo chmod o+rx /var/www   # Other can read and `cd` into the directory (+rx)
```

These permissions could also be set a bit more succinctly using the = operator:

```
sudo chmod ug=rwx /var/www
sudo chmod o=rx /var/www
```

This is useful if you have a user for deployment on your server. If a user is part of the group `www-data`, that user will now have permissions to add/update files within the `/var/www` directory.

Files created by a user belong to that user's username and group. This means that after creating/deploying files, we'll likely need to set file permissions properly.

For example, after a deployment to `/var/www`, the deployment user should set the group and permissions of the new and updated files. The files should be assigned the group `www-data` and have group read/write abilities set to them.

User Management

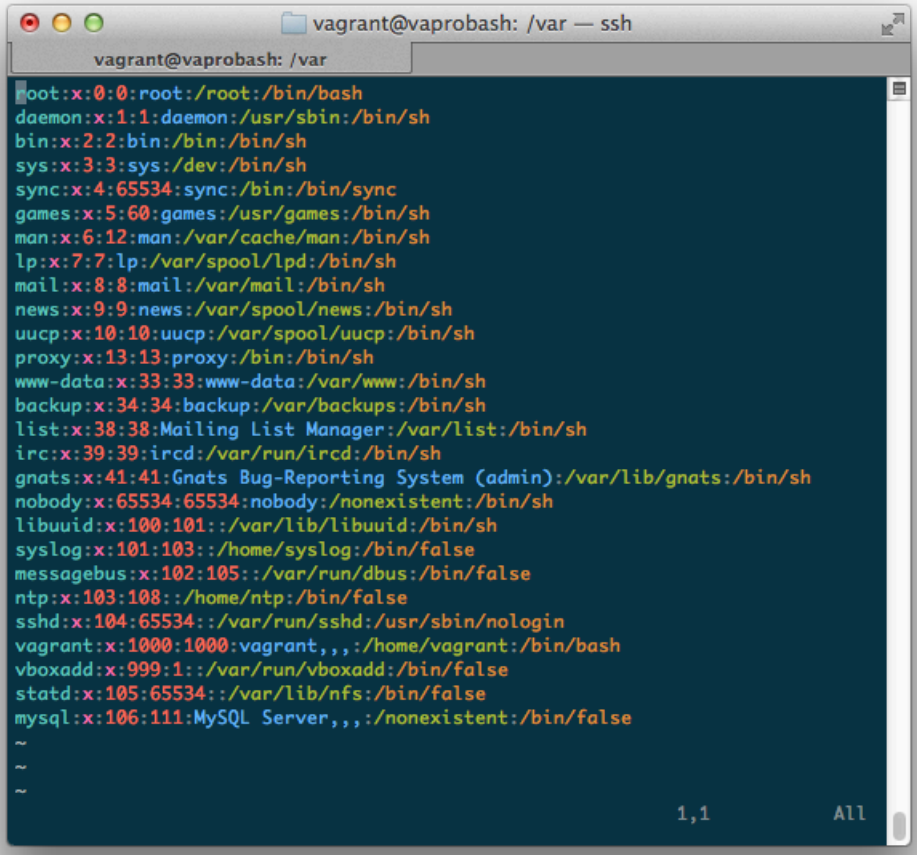
We need to also manage users and what groups they belong to.

Every user created by default belongs to a user and group of the same name. Users can belong to one primary group, and then can be added to many other secondary groups.

The primary group is usually what files/directories are assigned when a user creates a new file or directory. (Their username is of course the user assigned to those same files/directories).

We can find a list of users created in the `/etc/passwd` file:

```
1  vim /etc/passwd
```



/etc/password

This will show us the following information in colon-separated columns:

- User
- Password ("x" meaning the user has an encrypted password)
- User ID (UID)
- Group ID (GID)
- User Information (extraneous notes)
- Home Directory
- Command/Shell used by the user

For more information on this list, including some notes on the UID/GID's, see this article on understanding the /etc/passwd file format[17].

[17] http://www.cyberciti.biz/faq/understanding-etcpasswd-file-format/

Creating Users

Let's create a new user to use for deployments. We'll name this user "deployer".

```
sudo adduser deployer
```

Note that `adduser` is not the same as `useradd`, although both commands usually exist. The `adduser` command does some work which we'd have to otherwise do manually. Use the `adduser` command. You can remember which command to use by thinking of the word order used if saying "I want to add a user" out loud.

This will do some setup and ask you for a password for this user. This might also ask you for the user's full name and some other information.

If we check out `/etc/passwd` again, we'll see a new line similar to this:

```
deployer:x:1001:1003:,,,:/home/deployer:/bin/bash
```

In the above example, our user "deployer" has a UID of 1001 and GID of 1003.

We can act as this user by running the command:

```
sudo su - deployer
```

Then we can type in `groups` to see what groups we are part of:

```
$ groups
deployer
```

If you're following along, run the `exit` command to go back to your sudo user.

Let's set our `deployer` user to have a secondary group of `www-data`.

```
sudo usermod -a -G www-data deployer
```

We use `-a` to "append" the group to the users current secondary groups. The `-G` (upper-case "G") assigns the user `deployer` the group `www-data` as a secondary group.

If a directory or file is part of the `www-data` group and has group read-write permissions set, our user `deployer` can will be able to read and modify it. Our `deployer` user can deploy to `www-data` directories!

Alternatively, you can make your deploy user's primary group `www-data`. New files/directories created will then be part of group `www-data`.

To do so, run:

```
sudo usermod -g www-data deployer
```

The `-g` (lower-case "g") will assign the user `deployer` the group `www-data` as its *primary* group. Any files/directories created by this user will then have the `www-data` group assigned to it. We can then skip the step of changing the group permissions of files/directories after deployment.

Those steps, including creating a user and assigning it the primary group `www-data`, look like this:

```
sudo adduser deployer              # Fill in user info and password
sudo usermod -g www-data deployer  # Assign group www-data (primary)
```

Then we can make sure our web files are in group `www-data` and ensure group members have proper permissions:

```
sudo chgrp -R www-data /var/www
sudo chmod -R g+rwx /var/www
```

Umask & Group ID Bit

I've mentioned user and group permissions used for deployment often. We can simplify the use of group permissions by using umask and group id bits.

We'll do two things:

1. We will tell users to create new files and directories with group read, write and execute permissions.
2. We will ensure new files and directives created keep the group set by their parent directory

This will let us update files to our servers without having to reset permissions after each deployment.

Umask

First, we'll use umask to inform the system that new files and directories should be created with group read, write and execute permissions.

Many users have a umask of 022. These numbers follow the User, Group and Other scheme. The series 022 means:

- 0 - User can read, write and execute
- 2 - Group can read, execute
- 2 - Other can read, execute

Here's what octal values we can use for each of the three numbers:

- 0 - read, write and execute
- 1 - read and write
- 2 - read and execute
- 3 - read only
- 4 - write and execute
- 5 - write only
- 6 - execute only
- 7 - no permissions

In our setup, we want the group members to be able to write, not just read and execute. To do so, we'll set that to zero for user `deployer`:

```
sudo su - deployer
umask 002
```

Then any new directory will then have `g=rwx` permissions. New files will have `g=rw` permissions. Note this doesn't give execute permission to files.

The umask needs to be set for each user. You can use `sudo su - username` to change into any user and set their umask.

```
# Ensure user deployer is also part of group www-data
sudo usermod -a -G www-data deployer

# Set umask for user deployer
sudo su - deployer
umask 002

# Set umask for user www-data
sudo su - www-data
umask 002
```

You should also set this within the each user's `~/.bashrc`, `~/.profile`, `~/.bash_profile` or similar file read in by the users shell. This will then set the umask for the user every time they login.

File ~/.bashrc, the bash file read in Ubuntu for each user when logged into a shell

```
# Other items above omitted
umask 002
```

Then save and exit from that file. When you next login (or `source ~/.bashrc`) the umask will be set automatically. This works for when automating scripts run by certain users a well.

Group ID Bit

We've made users create files and directories/files with group write and execute permissions as applicable. Now we need new files/directories to take on the group of their parent directories. We can do this with the "group id bit".

We'll use a familiar command to do that:

```
sudo chgrp www-data /var/www # Change /var/www group to "www-data"
sudo chmod g+s /var/www # Set group id bit of directory /var/www
```

If you then inspect the `/var/www` directory, you'll see that in place:

```
$ ls -lah /var/www
total 12K
drwxrwsr-x  2 www-data www-data 4.0K Sep 13 17:58 .
drwxr-xr-x 14 root     root     4.0K Sep 13 17:54 ..
-rwxrw-r--  1 www-data www-data    6 Sep 13 17:58 index.html
```

New files created by user `www-data` or `deployer` will then be part of group `www-data` and maintain the proper group permissions!

This is a great setup for automated deployments. We can worry less about file permissions when automating deployments and other processes. You just need to remember to do the following:

- Set the umask for EVERY user of group `www-data` that might do file operations in the application files
- Set the correct group owner and add the `+s` group id bit for the proper directories

Running Processes

Processes (programs) are actually run as specific users and groups as well. This means we can regulate what processes can do to the system using file and directory permissions.

Core processes which need system access are often run as user root. Some run as user root, but then spawn processes as other users. This "downgrading" of privileges is used for security - we don't want PHP-FPM processes running PHP code as user root in a production server!

For example, Apache is started as user root. The master process then downgrades spawned processes to less privileged users. This lets Apache listen on port 80 (which requires root privileges) while reducing the harm spawned processes can do.

Webservers

As this book is mainly about what we need to know as web programmers, the sections on web servers are some of the most detailed.

First we'll see an overview of how servers match incoming HTTP requests to a website. Then we'll discuss the finer topics of installing, configuring and using Apache and Nginx.

We'll see how to integrate our applications with these web servers. Finally, we'll get in-depth with PHP-FPM.

HTTP, Web Servers and Web Sites

You likely know that a web server can handle serving more than one web site. In Apache, this is done by defining Virtual Hosts. In Nginx, this is done by defining Servers within the Nginx configuration (commonly also referred to Virtual Servers).

If a web server is hosting multiple web sites, how does the server route incoming requests to the correct web site?

It reads the HTTP request's `Host` header. If the Host header is not present or doesn't match a defined site, the web server routes the request to a default site.

We can see this in action using `curl`. For this example, we'll use two of my websites, fideloper.com and serversforhackers.com. These happen to exist on the same server as of this writing.

Let's get the IP address of the server:

```
$ ping fideloper.com
PING fideloper.com (198.211.113.202): 56 data bytes
```

So we can see the IP address is 198.211.113.202. Let's use `curl` to see what response we get when using the IP address only.

```
$ curl -I 198.211.113.202
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Mon, 16 Jun 2014 02:07:47 GMT
Content-Type: text/html
Content-Length: 178
Connection: keep-alive
Location: http://serversforhackers.com/
```

We can see that I'm using Nginx. Nginx responds with a 301 redirect response, telling the client to head to http://serversforhackers.com via the `Location` header. This means two things:

1. Serversforhackers.com is the default site, rather than fideloper.com
2. The Nginx configuration sends a 301 redirect to the domain serversforhackers.com. Some web servers might just serve the default content. I happen to have a 301 Redirect configured so the site is always accessed via a domain.

Let's next see what happens when we add a Host header to the HTTP request:

```
1  $ curl -I -H "Host: fideloper.com" 198.211.113.202
2  HTTP/1.1 200 OK
3  Server: nginx
4  Content-Type: text/html; charset=UTF-8
5  Connection: keep-alive
6  Vary: Accept-Encoding
7  Cache-Control: max-age=86400, public
8  Date: Mon, 16 Jun 2014 02:10:34 GMT
9  Last-Modified: Fri, 09 May 2014 20:54:31 GMT
10 X-Frame-Options: SAMEORIGIN
11 Set-Cookie: laravel_session=somerandombits
```

We can see a Laravel session created on line 11. Fideloper.com happens to be a Laravel based web application. We can infer that this request was routed to the fideloper.com site successfully!

Next, we'll request the Servers for Hackers site on the same server:

```
1  $ curl -I -H "Host: serversforhackers.com" 198.211.113.202
2  HTTP/1.1 200 OK
3  Server: nginx
4  Date: Mon, 16 Jun 2014 02:13:10 GMT
5  Content-Type: text/html; charset=utf-8
6  Content-Length: 5943
7  Last-Modified: Mon, 02 Jun 2014 23:32:32 GMT
8  Connection: keep-alive
9  Vary: Accept-Encoding
10 Expires: Mon, 16 Jun 2014 02:13:09 GMT
11 Cache-Control: no-cache
12 X-UA-Compatible: IE=Edge,chrome=1
13 Accept-Ranges: bytes
```

Here we see a different set of headers with no session created. We've reached serversforhackers.com! This is a static site built with the site generator Sculpin.

So, we can see how the `Host` header is used by the web server to determine which website to direct a request to. In the next chapters, we'll cover installing and configuring Apache and Nginx for one or more websites.

A Quick Note on DNS

We've seen that the Host header can inform our web server what website a client requested. However, how does a domain used in a user's browser reach our server in the first place?

This is the job of DNS (Domain Name Service). When we purchase a domain from a registrar, we often also need to add some domain name servers for the domain. These DN entries will often look like `ns1.somehost.com` and `ns2.somehost.com`.

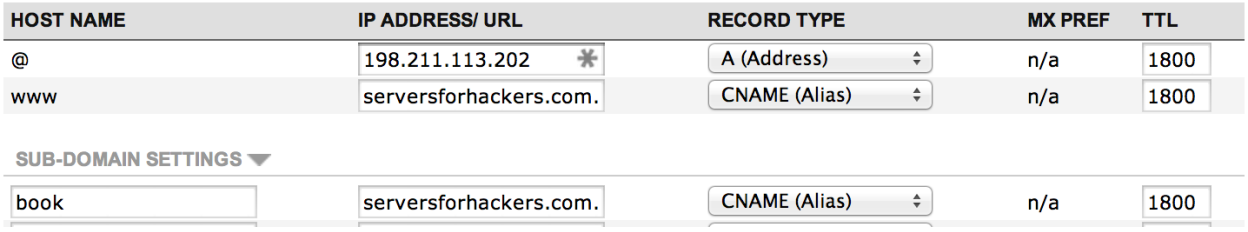
Many registrars setup their own domain name services for you. However, you can purchase a domain from a registrar but control your domain name information on another service! Registrars will allow you to set which Name Servers are used for your domain. You can set the Name Servers (`ns1.somehost.com` for example) to those of another service of your choice.

Some third-party domain name services to consider:

- AWS Route 53
- CloudFlare CDN (has DNS services)
- DynDNS
- OpenDNS
- EasyDNS
- DNSMadeEasy

These should all work pretty well. The only recommendation I have is to not use a DNS provided by your hosting. Some of the Name Servers of the popular cloud server providers have come under attack before. In this situation, the DNS services for your domain won't work even if your application servers are up and running fine. Using a separate DNS service is an easy way to not put your eggs all in one basket.

Once your Name Servers are set, you can head to your registrar or DNS service of choice and start adding DNS records. This is where you can point your domain to your web server. Then once someone uses your domain in the browser, it will be directed to your web server!

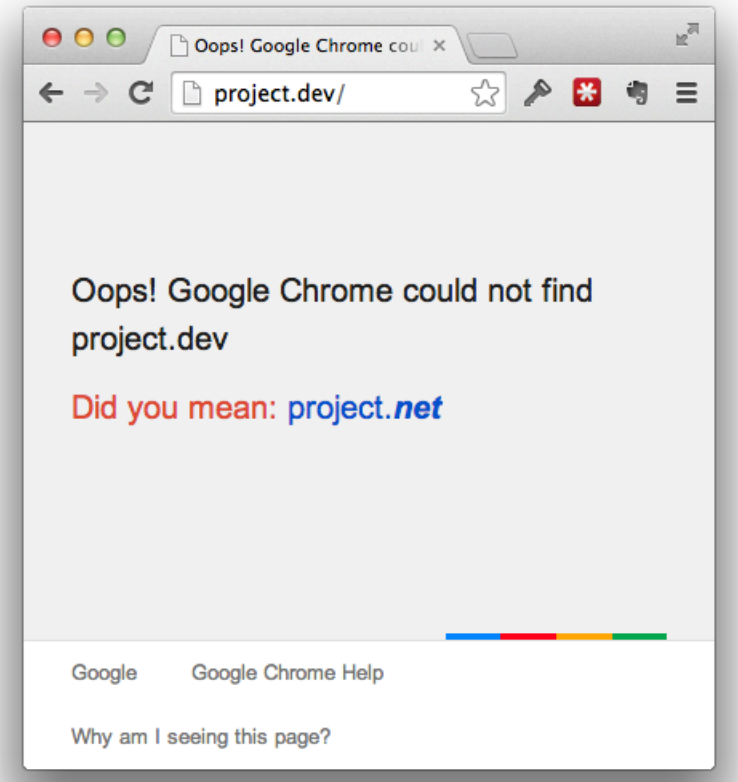


DNS entries for serversforhackers.com

DNS & Hosts File

Let's say you want to use a domain for development on your local server. You have a virtual machine (perhaps Vagrant) or some server on which you'll be developing. If you want to reach this server by using the domain `project.dev`, how would you accomplish that?

If use the URL `http://project.dev` in our browser, the browser won't know what to do with it. There's no mechanism in place to tell the browser what IP address the domain should resolve to.



project.dev before hosts file

What we need is a way to tell the browser that the domain `project.dev` points to some IP address. Luckily, all computers (in all OSes), have a `hosts` file. This file lets us map domains to IP addresses.

What does a hosts file look like? Many just have a few entries pointing `localhost` to your local IP address.

On my Macintosh, it looks like this:

File: /etc/hosts - localhost and ipv4 and ipv6 addresses

```
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1   localhost
255.255.255.255 broadcasthost
::1         localhost
fe80::1%lo0 localhost
```

In Windows, the location of the hosts file is usually `%systemroot%\system32\drivers\etc\`. `%systemroot%` is often `C:\Windows`.

Let's say our development server is on the IP address `192.168.22.10`. If we want to use the hostname `project.dev` to reach this server, then we can append this entry to the hosts file:

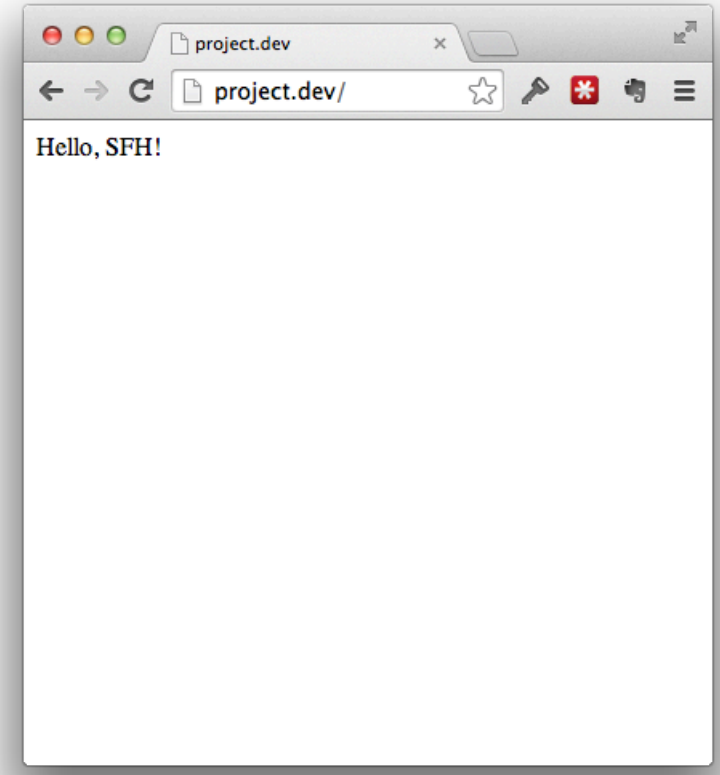
```
192.168.22.10 project.dev
```

We can add multiple domains to the same entry as well. This will point each domain to the same IP address:

```
192.168.22.10 project.dev project2.dev codename-orange.dev another.domain.dev
```

After saving your hosts file, you'll find these domains start to work. You can enter them into your browser or use them in SSH connections. The domains will resolve to the IP address you set!

These changes will *only* work on the computer whose `hosts` file you edited. It's a local modification only.



project.dev after hosts file

Xip.io

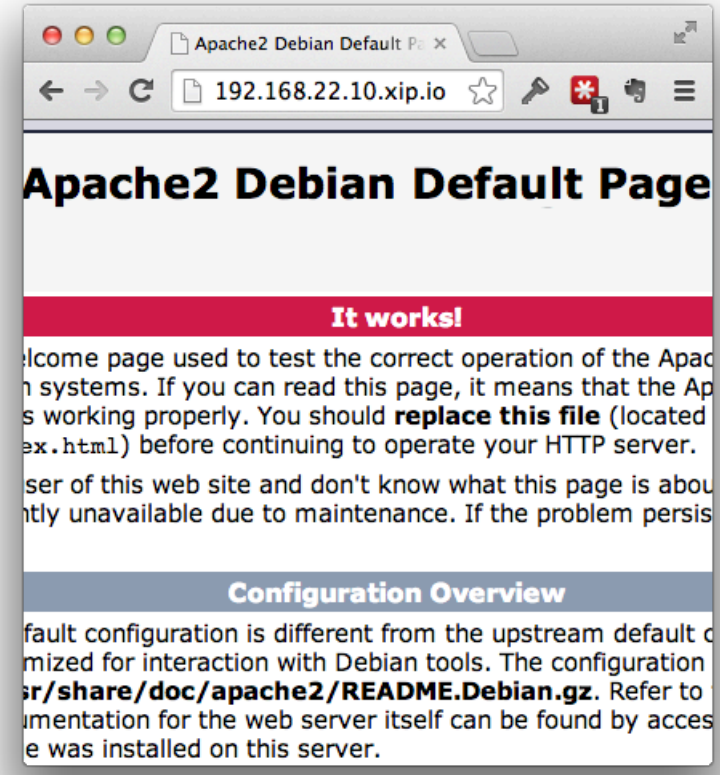
So what is our hosts file doing for us here? It's enabling us to circumvent the need to have any DNS services enabled for our domain. In other words, we don't need to setup our own DNS service. For our each local computer, we can edit the hosts file and be done with it.

The hosts file is providing the service of telling our computer what IP address to resolve to when the domain is used.

But there are services available which will let us skip having to edit our hosts file. Xip.io[18] is such a service; It can act like your hosts file. If you specify the IP address you'd like the `xip.io` domain to resolve to, it will do that for you!

For example, if I skip editing my hosts file and instead use `192.168.22.10.xip.io` in the browser, that will reach the server as well!

[18] http://xip.io



xip.io

The xip.io service saw the IP address in the subdomain(s) of the `xip.io` address and resolved to it.

We can even use more subdomains with `xip.io`! For example, `whatever.i.want.192.168.22.10.xip.io` will work just as well. This lets us use subdomains to differentiate between our projects, if they are hosted on the same server.

Using xip.io *does* require internet access - something to keep in mind when developing on the go.

Virtual Hosts

Perhaps you noticed that the `xip.io` address landed us on the default Apache site page, instead of our project page. The second part of this is making sure the virtual hosts on our webserver know what to do with the domain given.

Do you see what's happening? Editing your hosts file points the domain to the correct server. However, your web server at that IP address still needs to know what to do with that web request!

The other half of the equation is making sure our Apache or Nginx virtualhost routes to the right site/webapp. It's not enough just to point a domain to our web server's IP address.

Web servers look for the `Host` header in an HTTP request to map the request to a configured website. Using `xip.io` will provide a `Host` header - we just need our web servers to know what to do with those requests.

On **Apache**, we can do that by editing our virtual host do something like this:

Fictitious virtual host file /etc/apache2/sites-available/project.dev.conf

```
<VirtualHost *:80>
        ServerName project.dev
        ServerAlias project.*.xip.io

        DocumentRoot /vagrant/project
</VirtualHost>
```

What's this doing? Well we set the site's primary domain as `project.dev`, in case we want to use that instead of `xip.io`. If we want to make use if xip.io, we can use a wildcard in place of the IP address. We don't need to update the virtual host if our server's IP address changes. Note that we must set that up in the `ServerAlias`, as Apache's `ServerName` directive can't use wildcards.

For example, to match the domain `project.192.168.22.10.xip.io`, we use `project.*.xip.io`.

If we later want to have another project on the same server, we can create another virtual host:

Fictitious virtual host file /etc/apache2/sites-available/project-two.dev.conf

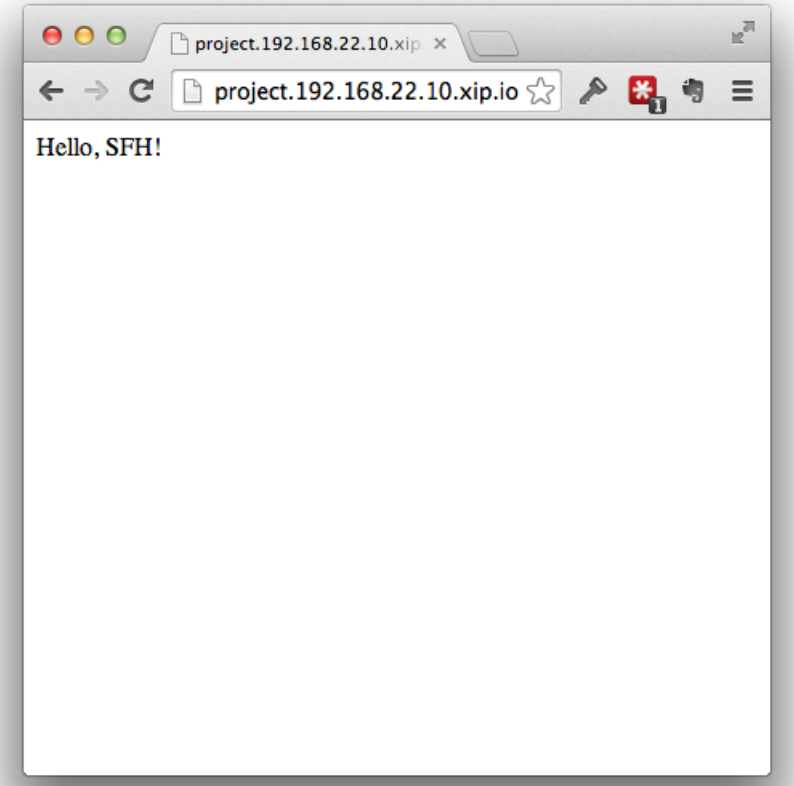
```
<VirtualHost *:80>
    ServerName project-two.dev
    ServerAlias project-two.*.xip.io

    DocumentRoot /vagrant/project-two
</VirtualHost>
```

In **Nginx**, we will do similarly:

Fictitious virtual host file /etc/nginx/sites-available/project.dev

```
server {
    listen: 80;

    server_name project.dev ~^project\.(.*)\.xip\.io;

    root /vagrant/project;

    index index.html index.htm;

    location / {
            try_files $uri $uri/ /index.html;
    }
}
```

Here we have a similar setup. Using the `server_name` directive, we can set up two domains. We have told it to respond to `project.dev`, as well as `project.*.xip.io`, using a regular expression.



wildcard xip.io

The **most important** point of the above steps is to know that simply pointing a domain to your server is not enough. In order for your web application to work, your web server also needs to know what to do with the incoming request.

You can use your hosts file to point real domains to another server. This is useful for testing an application as if it's "in production".

Hosting Web Applications

In the following chapters, we'll discuss configuring Apache and Nginx. Before we do, we should discuss how hosting a modern web application works.

Web frameworks of all languages include a way to run an application in the browser during development.

Things get more complicated when we want to host an application. We can't just run `python app.py` or `php -S 0.0.0.0:80` on a server used for real traffic and hope for the best!

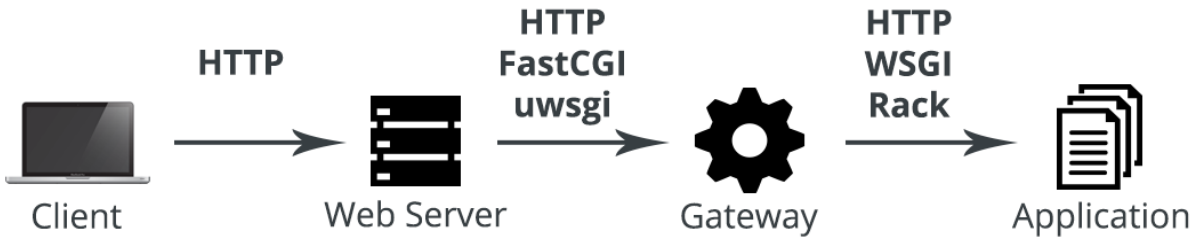
A "real server" in this context is a production server. However it could be any remote server, whether for development, staging, or production.

Three Actors

Hosting a web application requires the orchestration of three actors:

1. The Application
2. The Gateway
3. The Web Server

Here's the general flow of a web request into an application. We'll discuss this flow going from right to left.



application gateway request flow

Applications & HTTP Interfaces

Web Applications are generally coded using a framework or suite of libraries. These typically have tooling to handle HTTP requests.

Libraries created to accept and translate HTTP requests are referred to as HTTP Interfaces. These accept requests and translate them for application code.

For example, Python has the WSGI specification. This specifies an interface between web servers and Python applications.

A popular implementation of WSGI is Werkzeug. This is a Python library that can accept and parse WSGI-compliant web requests.

Similarly, Ruby has Rack. Rack is a specification *and* library that can accept Rack-compliant web requests.

Most languages have HTTP interfaces available. Python and Ruby have HTTP interfaces added on via specifications and libraries. However, many newer languages include HTTP interfaces as part of their standard library.

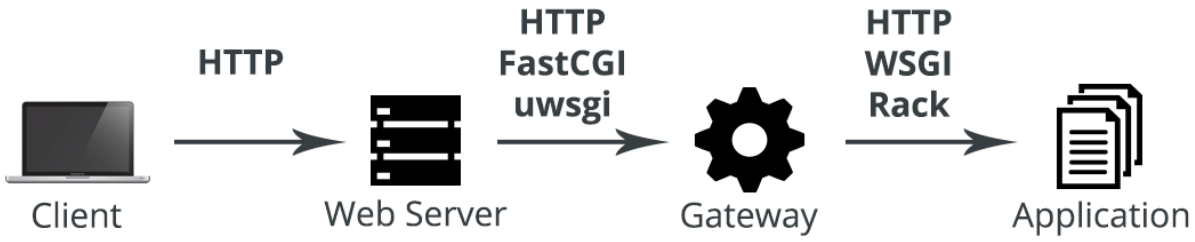
NodeJS and Golang are two examples of languages that can listen for HTTP requests "out of the box". HTTP requests can be given and parsed without needing libraries or specifications.

PHP, notably, is lacking such a specification. However, there are talks of PHP-FIG defining such an interface in PSR-7[19].

In any case, web applications must incorporate a way to accept web requests and return valid responses.

We've discussed how languages specify how to accept web requests. Next, we'll discuss how application gateways translate HTTP requests.

The application request flow and its interaction with a gateway is pictured here.



application gateway request flow

[19] https://github.com/php-fig/fig-standards/blob/master/proposed/http-message.md

The Gateway

Gateways sit between a web server (Apache, Nginx) and a web application. They accept requests from a web server and translate them for a web application.

Unfortunately, gateways typically don't label themselves as such. The exact definition of a gateway is somewhat fluid.

Some call themselves HTTP servers. Some consider themselves process managers. Others are more of a platform, supporting multiple use cases, protocols, and programming languages.

It might be useful to describe what gateways do rather than pin down an exact definition. Some common functionality of gateways include:

1. Listen for requests (HTTP, FastCGI, uWSGI and more)
2. Translate requests to application code
3. Spawn multiple processes and/or threads of applications
4. Monitor spawned processes
5. Load balance requests between processes
6. Reporting/logging

A gateway's main purpose is usually to translate requests. It's also common for a gateway to control application processes and threads.

We'll concentrate on the translation of requests.

Consider a gateway receiving a request meant for a Python application. The gateway will translate the request into a WSGI-compliant request.

It's the same for a gateway receiving a request for a Rack application. The way gateway will translate the request into a rack-compliant request.

Of course, in order for a gateway to translate a request, they must first receive one.

PHP-FPM, the gateway for PHP, is an implementation of FastCGI. It will listen for FastCGI requests from a web server.

Many gateways can accept HTTP requests directly. uWSGI, Gunicorn, and Unicorn are examples of such gateways.

Other protocols are also often supported. For example, uWSGI will accept HTTP, FastCGI and uwsgi (lowercase, the protocol) requests.

Don't confuse Python's PEP 3333 (WSGI) specification[20] with uWSGI's protocol "uwsgi[21]".

WSGI is Python specification for handling web requests. uWSGI can translate a request to be compatible with WSGI applications.

Similarly, uWSGI has it's own specification called "uwsgi". This specifies how other clients (web servers) can communicate with uWSGI!

A web server such as Nginx can "speak" uwsgi in order to communicate with the gateway uWSGI. uWSGI, in turn, can translate that request to WSGI in order to communicate with an application. The application will accept the WSGI-compliant request for an application. The Werkzeug library is capable of reading such a WSGI request.

No matter what protocol is used, gateways can accept a request and translate it to speak a web application's "language".

The following gateways will translate requests for WSGI (Python) applications:

- Gunicorn
- Tornado
- Gevent
- Twisted Web
- uWSGI

The following gateways will translate requests to Rack (Ruby) applications:

- Unicorn
- Phusion Passenger
- Thin
- Puma

A modern way to run PHP applications is to use the PHP-FPM gateway. PHP-FPM will listens for FastCGI connections.

Users of HHVM can use the included FastCGI server to listen for web requests. It acts much like PHP-FPM.

Before PHP-FPM, PHP was commonly run *directly* in Apache. A Gateway was not used. Instead, Apache's PHP module loaded PHP directly, allowing PHP to be run inline of any files processed.

This is still a common way to run PHP.

[20] http://legacy.python.org/dev/peps/pep-3333/

[21] http://uwsgi-docs.readthedocs.org/en/latest/Protocol.html

Skipping Gateways

I mentioned above that some languages include HTTP interfaces in their standard library.

Applications built in such languages *can* skip the use of gateways. In that scenario, a web server will send HTTP requests directly to the application.

Such applications can still benefit from the use of a gateway. For example, NodeJS applications. Node's asynchronous model allows it to run efficiently as a single-process. However, you may want to use multiple processes on multi-core servers.

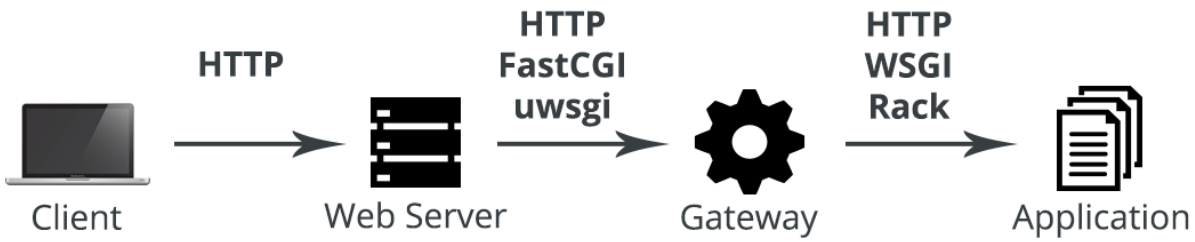
A NodeJS gateway such as PM2[22] could manage multiple processes. This would allow for more concurrent application requests to be handled.

Gateways aren't necessarily language-specific! For example, uWSGI, Gunicorn and Unicorn have all been used with applications of various languages.

You'll find that tutorials often match a gateway with applications of a specific language. This isn't a hard rule. In fact, uWSGI is written in C rather than Python!

This is why specifications exist. They allow for language-agnostic implementations.

The gateway request flow described is pictured here. We've discussed the flow from a web server to the gateway and from the gateway to the application.



application gateway request flow

The Web Server

Web servers excel at serving requested files, but usually serve other purposes as well.

Popular web-server features include:

- Hosting *multiple* sites
- Serving static files

[22] https://github.com/Unitech/pm2

- Proxying requests to other processes
- Load balancing
- HTTP caching
- Streaming media

Here we're concerned with the web server's ability to act as a (reverse) proxy.

The web server and the gateway are middlemen between the client and an application. The web server accepts a request and relays it to a gateway, which in turn translates it to the application. The response is relayed back, finally reaching the client.

We've briefly discussed how a gateway can accept a request and translate it for an application. We'll get in a little more detail here.

As mentioned, a web server will translate an HTTP request to something a gateway can understand. Gateways listen for requests using various protocols.

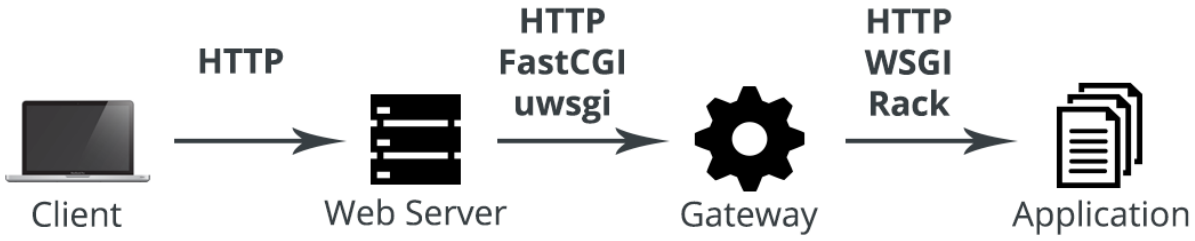
Some gateways can listen for HTTP connections. In this case, the web server can relay the HTTP request to the gateway directly.

Other gateways listen for FastCGI or uwsgi connections. Web servers which support these protocols must translate an HTTP request to those protocols.

Nginx and Apache can both "speak" HTTP, uwsgi, and FastCGI. These web servers will accept an HTTP request and relay them to a gateway in whichever protocol the gateway needs.

More specifically, the web server will translate a request into whatever you configure it to use. It's up to the developer/sysadmin to configure the web server correctly.

The web server request flow described is the flow from the client to the web server and from the web server to the gateway.



application gateway request flow

PHP is Special

PHP is unique in that it's a language built specifically for the web.

Most other languages are either general purpose or do not concentrate on the web. As a result, PHP is fairly different in how it handles HTTP requests.

PHP was originally built under the assumption that it is run during an HTTP request. It contained no process for converting the bytes of an HTTP request into something for code to handle. By the time the code was run, that was dealt with.

PHP is no longer limited to being run within context of an HTTP request. However, running PHP outside of a web request was an evolution of PHP, not a starting point.

Conversely, other languages have a process of translating an HTTP request into code. This usually means parsing the bytes constituting HTTP request data.

Many libraries have been created as a language add-on to handle HTTP requests. Some newer languages can handle HTTP requests directly. These languages don't assume code is always run in the context of an HTTP request, however.

This process in PHP is roughly equivalent to using cURL (or perhaps the Guzzle package) to accept web requests.

On a practical level, this means that PHP's super globals (`$_SERVER`, `$_GET`, `$_POST`, `$_SESSION`, `$_COOKIE` and so on) are already populated by the time your PHP code is run. PHP doesn't need to do work to get this data.

PHP-FPM, the gateway for PHP, takes a web request's data and fills in the PHP super globals. It sets up the state of the request (the environment) before running code.

PHP never needed a specification or library for accepting bytes of data and converting it to a request. It's already done by the time PHP code is run! This is great for simplicity.

Modern applications, however, are not simple. More structured applications ("Enterprise") often need to deal with HTTP in great detail. Perhaps we want to encrypt cookies or set cache headers.

This often requires us to write objects *describing* HTTP. These objects are populated by PHP's environment/state. These objects then adjust HTTP-related state, applying correct HTTP business logic and constraints.

Under the hood, such libraries use PHP's built in HTTP-related functions to read in a request and create a response.

Popular examples of these libraries includes Symfony's HTTP libraries, and the possible PSR-7 standard.

An HTTP interface can help standardize and encapsulate HTTP concerns as well. This is great for testing. Accessing and modifying super globals (global state) can often lead to issues in testing and code quality.

For the end-user (developer), that's a hidden difference between PHP and other languages.

Languages like Python and Ruby have a specification for how raw HTTP information should be sent. It's up to frameworks and libraries to handle HTTP concerns. Code manipulating HTTP response data must be written to send responses in the proper HTTP format.

PHP frameworks simply manipulate HTTP request state given. For HTTP responses, PHP has built-in methods to let you adjust HTTP headers and data.

PHP developers don't need to concern themselves with how data gets translated into a HTTP response. Instead, that's a concern for the internals team.

Apache

First we'll cover the venerable Apache web server. Apache was, until very recently, considered the most popular web server used. By some counts, Nginx has recently taken the crown. In any case, Apache is still *very* widely used, making it worth learning about.

In this chapter, we'll look at using Apache as a basic web server, including setting up virtual hosts. Then we'll see a few ways to use it with some modern application languages, including (but not limited to) PHP and Python.

Installing

Before you install Apache, log into your server and try this command:

```
# Send an http request to "localhost"
# -I flag shows response headers only
$ curl -I localhost
curl: (7) Failed connect to localhost:80; Connection
```

If you don't have any web server installed, you should receive the error show above. That means there's nothing listening on the localhost network on port 80. Let's make this work!

On Debian/Ubuntu servers, the Apache package is called "apache2". Installing it is generally as simple as this:

```
sudo apt-get install apache2
```

I recommend using the ondrej/apache2[23] repository to keep up with the latest stable releases of Apache:

```
sudo add-apt-repository -y ppa:ondrej/apache2
sudo apt-get update
sudo apt-get install -y apache2
```

After installation, if you re-run the curl command, you should see a 200 OK Response in the headers sent back from Apache:

[23] https://launchpad.net/~ondrej/+archive/apache2

```
$ curl -I localhost
HTTP/1.1 200 OK
Date: Sun, 22 Jun 2014 13:22:43 GMT
Server: Apache/2.4.10 (Ubuntu)
Last-Modified: Sun, 22 Jun 2014 13:22:14 GMT
ETag: "2cf6-4fc6c9d7068b7"
Accept-Ranges: bytes
Content-Length: 11510
Vary: Accept-Encoding
Content-Type: text/html
```

Great, Apache is installed! The latest stable release is version 2.4.10 as of this writing. Let's move onto configuring some websites.

Configuration

In Ubuntu, Apache follows a common configuration scheme of `available` and `enabled` directories. Let's look at some Apache configuration directories:

- `/etc/apache2/conf-available`
- `/etc/apache2/conf-enabled`
- `/etc/apache2/mods-available`
- `/etc/apache2/mods-enabled`
- `/etc/apache2/sites-available`
- `/etc/apache2/sites-enabled`

We have available configuration files in the "available" directories. To enable an available configuration just place them in the corresponding "enabled" directory.

In practice, these configurations are enabled by creating a symlink ("symbolic link" aka an alias). That way we don't have to copy real files to the "enabled" directories - we can just create and destroy symlinks.

For example, if we have a site configured in `/etc/apache2/sites-available/001-mysite.conf`, we'll enable it by symlinking that to `/etc/apache2/sites-enabled/001-mysite.conf`. Then we can tell Apache to reload its configuration to read that new site in.

Checking the sites-available and sites-enabled directories

```
# Sites configured in sites-available
$ cd /etc/apache2
$ ls -la sites-available/
root root Jun 22 18:28 .
root root Jun 22 13:33 ..
root root Jan  7 13:23 000-default.conf
root root Jun 22 18:28 001-mysite.conf
root root Jan  7 13:23 default-ssl.conf

# Sites enabled in sites-enabled
# Note how the enabled sites are "pointing" to the ones we want enabled
# from the sites-available directory
$ ls -la sites-enabled/
root root Jun 22 18:29 .
root root Jun 22 13:33 ..
root root Jun 22 13:22 000-default.conf -> ../sites-available/000-default.conf
root root Jun 22 18:29 001-example.conf -> ../sites-available/001-mysite.conf
```

To enable a site configuration, create a symlink between an "available" and "enabled" directory:

Enabling a virtual host by creating a symlink in the sites-enabled directory

```
# Create a symlink between the actual conf in sites-available to the
# alias inside of sites-enabled:
sudo ln -s /etc/apache2/sites-available/001-mysite.conf \
           /etc/apache2/sites-enabled/001-mysite.conf

# Then reload Apache's configuration:
sudo service apache2 reload
```

Why the numbers in the filenames?

Virtual Hosts are processed in the order they appear in configuration. The *first matching* ServerName or ServerAlias determines the Virtual Host that is used. This is regardless of any wildcard domains defined.

The files are loaded in alpha-numeric order based on their filename. Because the first matching virtual host is used to serve a request, there are situations where we can use that to our advantage.

Alternatively, we can use the Apache tools `a2ensite` and `a2dissite` to enable a configuration:

```
# Enable a site
sudo a2ensite 001-mysite
sudo service apache2 reload

# Disable a site
sudo a2dissite 001-mysite
sudo service apache2 reload
```

Apache2 Tools

The following tools exist on Debian/Ubuntu to help with managing Apache configuration:

- `a2ensite` / `a2dissite` - Enable and disable virtualhosts by symlinking between `sites-available` and `sites-enabled`
- `a2enmod` / `a2dismod` - Enable and disable modules by symlinking between `mods-available` and `mods-enabled`
- `a2enconf` / `a2disconf` - Enable and disable modules by symlinking between `conf-available` and `conf-enabled`

These are not necessarily available on other Linux distributions.

To see how the configurations are loaded, let's inspect the main configuration file, `/etc/apache2/apache2.conf`:

Selections from /etc/apache2/apache2.conf

```
# Include module configuration:
IncludeOptional mods-enabled/*.load
IncludeOptional mods-enabled/*.conf

...

# Include list of ports to listen on
Include ports.conf

...

# Include generic snippets of statements
IncludeOptional conf-enabled/*.conf

# Include the virtual host configurations:
IncludeOptional sites-enabled/*.conf
```

This will load any configuration from the "*-enabled" directories. The configurations should end in ".conf" (or ".load" in the case of some modules).

The `Include` and `IncludeOptional` directives use wildcards. These will load files in alpha-numeric order. We can use filenames to ensure load order.

A specific load order will usually not be required. However, it may be useful when defining many virtual hosts. Complex configurations may depend on load order to load the correct site.

The enabled/available configuration convention is very useful. We can enable and disable of configuration without having to delete files!

We'll find this convention used commonly in the Debian/Ubuntu world. For example, it's used in Apache, Nginx and PHP!

Virtual Hosts

Apache uses "Virtual Hosts" to setup and configure multiple websites. Each website hosted on a web server can and should have their own Virtual Host configuration.

Virtual Hosts can be matched based on IP address or hostname.

The phrase "Virtual Hosts" is used a lot. They'll be referred to as a "vhost" from here on, just like in the official Apache documentation.

IP-Based Virtual Hosts

IP-based vhosts are configured per unique IP address and port combination. If a server has multiple public IP addresses assigned to it, we can set up a site per IP address on the same ports.

Let's say our server has these three fictitious IP addresses assigned to it: 123.123.123.111 through 123.123.123.113. In order to setup vhosts for all three IP addresses, we need to setup Apache to listen on them.

Most servers will only have one public IP address assigned to them. That's not always the case, however. One common reason to add extra IP addresses is when using an SSL certificate. In some situations, an IP address must be unique per domain under an SSL certificate. If a server has multiple sites using their own SSL certificates, they'll need more than one IP address.

You can get around the requirement for unique IP addresses per domain when using an SSL certificate. In fact this might be installed by default using Debian/Ubuntu's Apache2 package. Read more here[24].

[24] https://wiki.apache.org/httpd/NameBasedSSLVHostsWithSNI

To make Apache listen on our three IP addresses, edit the main configuration file `/etc/apache2/ports.conf`. Upon opening that file, you'll likely see something like this:

File: /etc/apache2/ports.conf

```
Listen 80

<IfModule ssl_module>
        Listen 443
</IfModule>

<IfModule mod_gnutls.c>
        Listen 443
</IfModule>
```

This sets Apache to listen on port 80 and 443 on **all** network interfaces the server is connected to. If you have reason to only listen on specific IP addresses, you can manually add `Listen` directives:

```
Listen 123.123.123.111:80
Listen 123.123.123.111:443
Listen 123.123.123.112:80
Listen 123.123.123.112:443
Listen 123.123.123.113:80
Listen 123.123.123.113:443
```

This will listen on both port 80 (http) and 443 (https) ports for the three example IP addresses.

This is not necessarily a common setup. Your webserver will likely need no such addition unless you want Apache to only listen on specific networks.

If you have made changes to the `ports.conf` file, close it and restart Apache:

```
sudo service apache2 restart
```

Once we have Apache listening on our IP addresses, we can setup a vhost for any of them as we need. The following shows a vhost declaration for a website at IP address 123.123.123.111 listening on port 80.

We might find this in file `/etc/apache/sites-available/example.com.conf`:

```
<VirtualHost 123.123.123.111:80>
        DocumentRoot /var/www/example.com/public
</VirtualHost>
```

IP-based vhosts are unique per IP address and port combination. If we need to listen to another IP address, we'll create another vhost:

```
<VirtualHost 123.123.123.112:80>
        DocumentRoot /var/www/another.example.com/public
</VirtualHost>
```

Note that the `DocumentRoot` directive tells Apache where the files for this website are.

Named-Based Virtual Hosts

IP-based vhosts are limited. Apache cannot have more than one vhost per IP address/port combination! Additionally, server hosts often charges for extra IP addresses.

Because of this, named-based vhosts are *far* more common.

Named-based vhosts work off of the hostname to match to a vhost. This hostname is taken from the `Host` header of an HTTP request. Let's see our `example.com` website based off of the hostname:

HTTP/1.0 does not include a `Host` header. An IP-based virtualhost may be required for the rare client which does not "speak" the newer HTTP/1.1.

File: /etc/apache2/sites-available/001-example.com.conf

```
<VirtualHost *:80>
        ServerName example.com
        ServerAlias www.example.com
        DocumentRoot /var/www/example.com/public
</VirtualHost>
```

In this case, we have some new/different directives:

- `*:80` - Tells Apache that the vhost listens on any IP address (any network interface) on port 80. Apache treats it as a named-based vhost when used with `ServerName`.
- `ServerName example.com` - Tell Apache what host to use to match to this website.
- `ServerAlias www.example.com` - Use the defined domains/hosts to also match these aliases. Often this can be the popular `www` subdomain. Multiple, space-separated hostnames can be defined.
- `DocumentRoot` - Tell Apache where the web files for this website are located on the file system.

Other Virtual Host Directives

Let's look at a common vhost setup for Apache and cover what each directive means. Again, this will be for `example.com`, with an example configuration file found at `/etc/apache2/sites-available/001-example.com.conf`.

File: /etc/apache2/sites-available/001-example.com.conf

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

We'll go over the options we see here:

Directive	Explanation
<VirtualHost *:80>	Listen on any network interface on port 80
ServerName	The hostname used to match an HTTP request's Host header to the vhost
ServerAlias	Alternate hostnames to use to match a request to a vhost. This can contain wildcards, and multiple hostnames can be used (space separated). Multiple ServerAlias directives can be used as well.
DocumentRoot	The directory path from where Apache should serve files
<Directory /file/path>	Apply given directives only for files (and sub-directories/files) in the given path.

Directive	Explanation
Options	Set available features within the server path. The "+" and "-" can enable or disable a feature.
Option: -Indexes	Do not display a directory listing of files in a directory if there's no index file (such as index.html, index.php). Removing this ability is more secure as users can't attempt to find files on your server by attempting to direct their browsers to directories. Use "+" instead of "-" to add the ability to see a directory index.
Option: +FollowSymLinks	Do follow symbolic links (aliases) if present.
Option: +MultiViews	Use mod_negotiation[25] to handle HTTP content negotiation.
AllowOverride All	Setting this to "ALL" allows the use of `.htaccess` files. Set this to `None` for `.htaccess` files to be ignored.
Require all granted	Apache 2.4+ changed access control to mod_authz_host[26]. This allows access to the web files to all. This used to be handled by a mix of "Allow" and "Deny" directives.
ErrorLog	Define an error log specifically for this vhost.
LogLevel	Define the verbosity of error log messages. Here, all messages of "warn" or of higher significance ("error", "crit", "alert", "emerg") are logged.
CustomLog	Part of mod_log_config[27], this lets you set an access log and optionally set a format. Above we use the "combined" log format.

Note that I added a second `ServerAlias` directive that matches the wild-carded hostname `example.*.xip.io`. This is useful for testing the virtualhost before making a site live.

In the above example, we didn't need to specify our servers IPs address within the vhost. This makes it easier if our server's IP address changes, as it may in development. We won't need to change our virtual host!

These are enough to get you up and running. In fact, they are just about all I use in most cases. Of course, your needs may vary. There are *many* more options to explore found in Apache's Core Features documentation[28].

Apache and Web Applications

Apache wouldn't be nearly so useful if we couldn't use it to send requests to web applications.

[25] http://httpd.apache.org/docs/current/mod/mod_negotiation.html
[26] http://httpd.apache.org/docs/2.4/mod/mod_authz_host.html
[27] http://httpd.apache.org/docs/current/mod/mod_log_config.html
[28] http://httpd.apache.org/docs/current/mod/core.html

To host a web application, a web server can accept an HTTP request and pass it (proxy it) off to a "gateway". The gateway handles converting the request into something an application can understand.

These gateways are various implementations and flavors of a "CGI"s - a Common Gateway Interfaces[29].

For example, many Python applications use the uWSGI[30] gateway. Apache will "proxy" a request to the gateway. In turn, the uWSGI gateway passes the request to the Python application.

PHP, when not directly loaded by Apache, can use the PHP-FPM gateway. FPM is an implementation of the FastCGI[31] gateway, which is a very common protocol.

Apache can also proxy to web applications over HTTP. This is popular when proxying requests to applications listening on HTTP. NodeJS and Golang are two languages that can listen for HTTP connections directly.

Gunicorn and Unicorn are two popular gateways which can communicate over HTTP as well. These can be used to serve Python and Ruby applications, respectively.

In the next sections, we'll discuss how Apache can talk to applications using HTTP, FastCGI and WSGI gateways.

Note that gateways are commonly tied to specific languages, but some are not!

Apache mod_php

Before we talk about commonly used gateways, let's discuss the glaring exception to the rule. PHP pages and applications are commonly loaded and parsed *directly* by Apache.

In this setup, Apache does not send PHP requests off to a gateway. Instead, Apache uses a PHP module to parse PHP requests directly. This allows PHP files to be used seamlessly alongside static web files.

Apache's `mod_php` makes using PHP extremely easy. It's commonly believed that this ease-of-use made PHP so successful. It is still commonly used.

Running the PHP module in Apache is as simple as installing Apache's `mod_php`. In Ubuntu, the package for that is "libapache2-mod-php5":

[29] http://en.wikipedia.org/wiki/Common_Gateway_Interface
[30] http://wsgi.readthedocs.org/en/latest/
[31] http://www.fastcgi.com/drupal/

```
sudo apt-get install -y libapache2-mod-php5
```

It's likely automatically enabled. However, you can ensure it's enabled by using the Debian/Ubuntu specific tool "a2enmod":

```
# Enable mod_php5
sudo a2enmod php5

# Restart Apache to load in the module
sudo service apache2 restart
```

What does "a2enmod" do? It simply creates the symlink for files `php5.load` and `php5-conf` files between the `mods-available` and `mods-enabled` directories. We could just as easily create the symlinks ourselves manually:

```
sudo ln -s /etc/apache2/mods-available/php5.load \
    /etc/apache2/mods-enabled/php5.load
sudo ln -s /etc/apache2/mods-available/php5.conf \
    /etc/apache2/mods-enabled/php5.conf

# Then restart Apache
sudo service apache2 restart
```

You can also use "a2dismod" to disable a module. Don't forget to restart Apache after disabling a module as well.

Apache should be restarted rather than reloaded after enabling/disabling modules.

Once the module is enabled and loaded, you can run PHP files in your websites without further configuration!

As we'll see, this is **NOT** actually a standard way to run a web application!

Going forward we'll see how to use Apache to send ("proxy") requests from Apache to various applications gateways.

Apache with HTTP

Apache can proxy requests to gateways or programs using HTTP. Some languages can speak HTTP directly while some gateways prefer to use HTTP.

Technically this means we could skip using a web server altogether and serve HTTP requests to them directly.

A more typical setup, however, is to put a web server such as Apache "in front of" an application. In such a setup, Apache would handle all HTTP requests. It would either handle the request itself or "proxy" the request to the application gateway.

This has certain benefits:

- Apache can handle requests for static assets. This frees the application from wasting resources on static assets.
- Apache can send requests to pools of resources of the application. Instead of one running NodeJS process, picture 3-4 running! Apache can send requests to each of them. This would substantially increase the number of requests the application could simultaneously handle. This essentially is load balancing.
- Some gateways monitor and manage multiple application processes for us. A gateway will expose one HTTP listener for Apache to send requests to. The gateway would then be responsible for sending requests to each running process. Some gateways can dynamically spin up and destroy running application processes.

Let's see how Apache can proxy requests off to an application listening for HTTP requests.

Here's an example NodeJS application. It will accept any HTTP request and respond with "Hello, World!".

File: /srv/http.js

```
#!/usr/bin/env node
var http = require('http');

function serve(ip, port)
{
        http.createServer(function (req, res) {
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end("Hello, World!\n");
        }).listen(port, ip);
        console.log('Server running at http://'+ip+':'+port+'/');
}

// Create a server listening on all networks
serve('0.0.0.0', 9000);
```

We can run this node "application" with the simple command: `nodejs /srv/http.js`. This application will listen on all network interfaces on port 9000. We can test this once it's running. You may need to open a new terminal window to test this while the NodeJS process is running:

```
# From within the server
$ curl localhost:9000
Hello, World!
```

Once that application is working, we need to configure Apache to send requests to it.

We're proxying requests directly to a test application. This NodeJS application is not a gateway. We'll see how to proxy requests to a gateway such as uWSGI or PHP-FPM in this chapter.

First we need to ensure the `proxy` and `proxy_http` modules are enabled. These allow Apache to proxy requests off to another process (application or gateway) over HTTP.

```
# Enable modules
sudo a2enmod proxy proxy_http

# Restart Apache
sudo service apache2 restart
```

Then we can adjust our vhost file to proxy requests off to our NodeJS application.

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    <Proxy *>
        Require all granted
    </Proxy>
    <Location />
        ProxyPass http://localhost:9000/
        ProxyPassReverse http://localhost:9000/
    </Location>
```

```
    <Location /static>
        ProxyPass !
    </Location>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

This vhost file is just like what we used earlier in this chapter. However there are some additions within the vhost. Let's cover those:

```
<Proxy *>
    Require all granted
</Proxy>
```

The Proxy[32] directive let's you configure all matched proxies. In this case, we're adding settings for for *all* configured proxies, as denoted by the wildcard '*'.

With the Proxy directive, we simply repeat `Require all granted`. This authorizes the request to be proxied from any host. This can be used to restrict what clients can use the proxy. In this case, we want the whole world to reach our application, and so we'll have Apache send requests to our application from any host.

Next we have a `<Location ...>` directive:

```
<Location />
    ProxyPass http://localhost:9000/
    ProxyPassReverse http://localhost:9000/
</Location>
```

The "location" represents a URI (and any sub-directory URIs) used. In this case, `/` effectively means "any URI". This will proxy all requests to the application by directing it to `localhost:9000`. Using `http` in the socket location tells Apache to proxy this as an HTTP request.

`ProxyPass` does the actual proxying while `ProxyPassReverse` tells Apache to adjust the `Location` and similar headers sent back from the proxy. For example if our application accidentally returns a header `Location: localhost:8080`, Apache can change that to the public-facing domain and port, perhaps `mysite.com:80`. The documentation[33] has some examples.

[32] http://httpd.apache.org/docs/current/mod/mod_proxy.html#proxy

[33] http://httpd.apache.org/docs/current/mod/mod_proxy.html#proxypassreverse

Lastly, we have a `<Location ...>` directive for the `/static` URI:

```
<Location /static>
    ProxyPass !
</Location>
```

We want Apache to handle requests for static assets. This is easy with PHP, whose files typically end in the `.php`. This allows us to pass requests ending in `.php` off to an application. We can say "Only send files ending in `.php` to the application".

Ths becomes an issue with other languages. Application of other languages typically don't run through a specific file.

One popular solution for informing Apache when to serve static assets is to put all static assets a specific directory. The above configuration does just that. Any URI which starts with `/static` will not be passed to the application. The `ProxyPass !` directive tells Apache not to proxy the request.

Apache will automatically add `X-Forwarded-*` headers[34] to servers when `ProxyPass` is used. More information about these headers and their use is in the Multi-Server Environments section of this book.

Multiple back-ends

We can proxy between multiple back-ends.

For example, let's pretend our application spawns multiple processes to listen on. This might be done to increase the number of concurrent requests it can handle. We'll simulate that by adjusting the last line of our NodeJS script to listen on three addresses:

File: /srv/http.js, bottom of file

```
serve('0.0.0.0', 9000);
serve('0.0.0.0', 9001);
serve('0.0.0.0', 9002);
```

Once edited, we can restart this process running `nodejs /srv/http.js`. It will then be listening on all network interfaces at port 9000, 9001 and 9002.

Next we can adjust our Apache configuration. We're essentially load balancing between the three back-end servers. To do so, we can use Apache's `proxy_balancer` module.

We also need to enable `lbmethod_byrequests`. This is the default method used by `proxy_balancer` to determine *how* Apache will balance between the back-ends.

[34] http://httpd.apache.org/docs/current/mod/mod_proxy.html#x-headers

You can find information on Apache's various Load Balancing algorithms in the `proxy_balancer` documentation[35]. "By Requests"[36] attempts to distribute traffic evenly amongst Apache workers.

We'll cover load balancing in depth in the Multi-Server Environments section. However we won't cover load balancing in Apache, as there are better and simpler tools.

```
# Enable the needed modules
sudo a2enmod proxy_balancer lbmethod_byrequests

# Restart Apache
sudo service apache2 restart
```

Then we can adjust the vhost file:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    <Proxy balancer://mycluster>
        BalancerMember http://localhost:9000/
        BalancerMember http://localhost:9001/
        BalancerMember http://localhost:9002/
    </Proxy>
    <Location />
        ProxyPass balancer://mycluster/
        ProxyPassReverse balancer://mycluster/
    </Location>
    <Location /static>
        ProxyPass !
    </Location>
```

[35] http://httpd.apache.org/docs/2.4/mod/mod_proxy_balancer.html
[36] http://httpd.apache.org/docs/2.4/mod/mod_lbmethod_byrequests.html

```
26
27      ErrorLog ${APACHE_LOG_DIR}/example.com-error.log
28
29      # Possible values include: debug, info, notice, warn, error, crit,
30      # alert, emerg.
31      LogLevel warn
32
33      CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined
34
35  </VirtualHost>
```

This setup is fairly similar. Let's go over the edited directives.

```
1  <Proxy balancer://mycluster>
2      BalancerMember http://localhost:9000/
3      BalancerMember http://localhost:9001/
4      BalancerMember http://localhost:9002/
5  </Proxy>
```

This defines a balancer cluster named 'mycluster'. The name can be anything. Then we define our three back-ends. In this case, the back-ends are the three Node HTTP listeners that we defined above.

Then our `<Location...>` directive needs tweaking to proxy requests to this balancer *cluster* rather than to the HTTP listener directly:

```
1  <Location />
2      ProxyPass balancer://mycluster/
3      ProxyPassReverse balancer://mycluster/
4  </Location>
```

That's it! This will distribute traffic amongst the three defined members of the balance cluster.

Our `<Location...>` directive for the `/static` directory is the same. It will continue to serve static assets.

The proxy module is handy for proxying to HTTP listeners. It can proxy requests to applications written in NodeJS or Golang. It's also commonly used to communicate with gateways listening on HTTP. Unicorn, Gunicorn and uWSGI are three common gateways which may create HTTP listeners.

Apache with FastCGI

Before Apache 2.4, we had to use `mod_fcgi` to send requests to a FastCGI gateway such as PHP-FPM. The `fcgi` module was nice in that once it was configured, you didn't have to worry about it again. However the configuration was needlessly complex.

As of Apache 2.4, we can use the `proxy_fcgi` module, which is much simpler!

In this section, we'll look at using `proxy_fcgi` via the `ProxyPassMatch` directive.

Then we'll look at how replacing `ProxyPassMatch` with `FilesMatch` can further simplify the configuration.

ProxyPassMatch

Let's see how to use `proxy_fcgi` to send PHP requests to the FastCGI gateway PHP-FPM.

First, we need to ensure the proper modules are enabled:

```
# Let's disable mod PHP first:
sudo a2dismod php5

# Then ensure mod_profyx_fcgi is enabled:
sudo a2enmod proxy proxy_fcgi

# Install PHP-FPM:
sudo apt-get install -y php5-fpm

# Restart Apache:
sudo service apache2 restart
```

Then we can edit our vhost to "proxy" to PHP-FPM FastCGI gateway, using the `ProxyPassMatch` directive. We'll edit the example configuration from the Virtual Host section above:

File: /etc/apache2/sites-available/001-example.conf

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
```

```
        AllowOverride All
        Require all granted
    </Directory>

    # THIS IS NEW!
    ProxyPassMatch ^/(.*\.php(/.*)?)$ \
        fcgi://127.0.0.1:9000/var/www/example/public/$1

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

We added the following line:

```
ProxyPassMatch ^/(.*\.php(/.*)?)$ fcgi://127.0.0.1:9000/var/www/example/$1
```

The `proxy_fcgi` module allows us to use `ProxyPassMatch` to match any request ending in `.php`. It then passes off the request to a Fast CGI process. In this case, that'll be PHP-FPM, which we'll configure to listen on the socket 127.0.0.1:9000. Note that we also pass it the file path where our PHP files are found. This is the same path as the `DocumentRoot`. Finally, we end it with `$1`, the matched PHP filename.

With Apache's traditional setup of using `mod_php`, we never had to worry about configuring Apache to serve PHP. Now we do - so any additional Virtual Host that may serve PHP files will need configuration for PHP.

Note that in proxying requests to PHP-FPM, we had to set the path to the PHP files. Unlike Nginx, Apache doesn't provide a DocumentRoot variable to pass to the `ProxyPassMatch` directive. This is unfortunate as it would have allowed for a more dynamic configuration with `ProxyPassMatch`.

Lastly we will reload Apache to read in the latest configuration changes:

```
sudo service apache2 reload
```

The last thing to do is edit PHP-FPM a bit. This will be covered fully in the PHP chapter, but we'll cover it briefly here. By default on Debian/Ubuntu, PHP-FPM listens on a Unix socket. We can see that in PHP-FPM's configuration file `/etc/php5/fpm/pool.d/www.conf`:

```
; The address on which to accept FastCGI requests.
; Valid syntaxes are:
;   'ip.add.re.ss:port'    - to listen on a TCP socket to a specific address on
;                            a specific port;
;   'port'                 - to listen on a TCP socket to all addresses on a
;                            specific port;
;   '/path/to/unix/socket' - to listen on a unix socket.
; Note: This value is mandatory.
listen = /var/run/php5-fpm.sock
```

We need to change this to listen on a TCP socket rather than a Unix one. Unfortunately `mod_proxy_fcgi` and the `ProxyPass`/`ProxyPassMatch` directives do **not** support Unix sockets.

```
# Change this from "listen = /var/run/php5-fpm.sock" to this:
listen = 127.0.0.1:9000
```

You can actually do this in this one-liner find and replace method:

```
sudo sed -i "s/listen =.*/listen = 127.0.0.1:9000/" /etc/php5/fpm/pool.d/www.conf
```

Lastly, as usual with any configuration change, we need to restart PHP-FPM:

```
sudo service php5-fpm restart
```

Once these are setup, files in that virtualhost ending in `.php` should work great!

Let's go over some pros and cons:

Pro:

- Works well out of the box with only minor configuration

Con:

- No Unix socket support. Unix sockets are slightly faster than TCP sockets, and are the default used in Debian/Ubuntu for PHP-FPM. Less configuration would be nice.
- `ProxyPassMatch` requires the document root set and maintained in the vhost configuration
- Matching non `.php` files takes more work. It's not so uncommon to see PHP inside of an `.html` file! This is also an issue when not using PHP - we need to pass in all URLs except for those of static files in that case.

FilesMatch

As of Apache 2.4.10, we can handle PHP requests with `FilesMatch` and `SetHandler`. This is a simpler but more solid configuration.

Apache 2.4.10 is recently released as of this writing. You can install version 2.4.10+ in Ubuntu by using the `ppa:ondrej/apache2` repository as described in the beginning of this chapter.

This still uses the `proxy_fcgi` module, so we need to ensure it's enabled once again:

```
# Let's disable mod PHP first,
# in case it's still on:
sudo a2dismod php5

# Then ensure mod_profyx_fcgi is enabled:
sudo a2enmod proxy_fcgi

# Install PHP-FPM if necessary:
sudo apt-get install -y php5-fpm

# Restart Apache:
sudo service apache2 restart
```

Then we can edit our Apache configuration. If you have a `ProxyPassMatch` line in there, comment it out or delete it.

Then, still in our example file:

File: /etc/apache2/sites-available/001-example.conf

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>
```

```
13
14      <FilesMatch \.php$>
15          SetHandler "proxy:fcgi://127.0.0.1:9000"
16      </FilesMatch>
17
18      ErrorLog ${APACHE_LOG_DIR}/example.com-error.log
19
20      # Possible values include: debug, info, notice, warn, error, crit,
21      # alert, emerg.
22      LogLevel warn
23
24      CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined
25
26  </VirtualHost>
```

The new directive here is:

```
1  <FilesMatch \.php$>
2      SetHandler "proxy:fcgi://127.0.0.1:9000"
3  </FilesMatch>
```

This matches any file ending in `.php` and then proxies the request off to PHP-FPM, using a TCP socket. If we elect to keep PHP-FPM on its default Unix socket, this directive now supports that as well:

```
1  <FilesMatch \.php$>
2      SetHandler "proxy:unix:/var/run/php5-fpm.sock|fcgi:"
3  </FilesMatch>
```

We can use this for proxying requests to any FastCGI gateway.

Let's cover what's different here from `ProxyPassMatch`:

First and foremost, we don't need to tell the handler where the PHP files are - this is agnostic of what the document root of a website is. This means the configuration is a bit more dynamic.

In fact, we could make this a global configuration. To do so, create a new file in `/etc/apache2`. I'll call it `php-fpm.conf`:

File: /etc/apache2/php-fpm.conf

```
<FilesMatch \.php$>
    # If using a Unix socket
    # Change this "proxy:unix:/path/to/fpm.socket|fcgi:"
    SetHandler "proxy:fcgi://127.0.0.1:9000"
</FilesMatch>
```

Once that file is created, you can include it within any Virtual Host configuration you'd like to use PHP:

File: /etc/apache2/sites-available/001-example.conf

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    Include php-fpm.conf

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

The line `Include php-fpm.conf` simply includes the `php-fpm.conf` file we created. We now have a configuration file we can selectively include into any vhost to pass requests to the FastCGI gateway PHP-FPM.

Note that this still uses RegEx to match files ending in `.php`. If we want to parse HTML files with php in it, we need RegEx to match PHP or HTML file extensions for the `FilesMatch` directive to proxy pass the request to PHP-FPM.

Lastly, note that I include the `php-fpm.conf` file "out in the open" of the vhost file. To add some security, we can apply this to only function within the DocumentRoot and its sub-directories. To do so, move the `Include` line inside of the `Directory` block.

Instead of:

```
<Directory /var/www/example.com/public>
    Options -Indexes +FollowSymLinks +MultiViews
    AllowOverride All
    Require all granted
</Directory>

Include php-fpm.conf
```

We would instead have:

```
<Directory /var/www/example.com/public>
    Options -Indexes +FollowSymLinks +MultiViews
    AllowOverride All
    Require all granted

    Include php-fpm.conf
</Directory>
```

We can include the `php-fpm.conf` file within the `Directory` block only because `FilesMatch` works within a `Directory` block. This is not the case for all configurations.

For example, `Proxy` and `Location` blocks cannot be places inside of a `Directory` block.

The Apache documentation will let us know what "context" a directive can be used. For example, we can see here that FilesMatch works in context of a `Directory` block[37].

So, in summary, using `FilesMatch` gives us these benefits:

- Not needing to define the DocumentRoot allows us to create a re-usable configuration
- We can use both Unix and TCP sockets

[37] http://httpd.apache.org/docs/2.4/mod/core.html#filesmatch

And these cons:

- We still need to do extra work to parse PHP in files not ending in `.php`
- If we're not using PHP-FPM, we need to capture all requests but those for static files

This is the method I use if using Apache with PHP-FPM.

Location

If we're **not** using PHP, then we can't really use `FilesMatch`, as we don't have a file to match most URI's to. PHP applications typically route all requests to an `index.php` file. However, most applications of other languages don't have any such file.

In these cases, we need to match against a directory-style URI instead of a file. We can do this exactly like we did with the HTTP proxy described above, using the `Location` block.

We still require the `proxy` and `proxy_fcgi` modules to proxy to FastCGI.

Enabling only the `proxy_fcgi` module will implicitly enable the `proxy` module.

Enabling the proxy_fcgi module and, implicitly, the proxy module

```
# Then ensure mod_profyx_fcgi is enabled:
sudo a2enmod proxy_fcgi

# Restart Apache:
sudo service apache2 restart
```

The Apache configuration is very similar to proxying to an HTTP listener as well - we just use the fcgi protocol instead!

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
```

```
        Require all granted
    </Directory>

    <Proxy *>
        Require all granted
    </Proxy>
    <Location />
        ProxyPass fcgi://127.0.0.1:9000/
        ProxyPassReverse fcgi://127.0.0.1:9000/
    </Location>
    <Location /static>
        ProxyPass !
    </Location>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

I'll cover the relevant portions quickly as they are basically the same as proxying to an HTTP listener.

The `<Proxy *>` blocks allows access to the proxy from all hosts (all web visitors to our site).

The `<Location />` block is used to accept all requests and proxy them to the FastCGI process listening at 127.0.0.1:9000.

If we used a Unix socket over a TCP socket, this would look the following:

```
<Location />
    ProxyPass unix:/path/to/socket.sock|fcgi:
    ProxyPassReverse unix:/path/to/socket.sock|fcgi:
</Location>
```

Finally the `<Location /static>` block is what we'll use to serve static content. This configuration assumes there's a directory named `/static`. It informs Apache that URI which starts with `/static` will be served directly rather than proxied. The `ProxyPass !` directive tells Apache not to proxy the request.

Multiple back-ends We may have multiple FastCGI back-ends as well. Again, just like with proxying to HTTP listeners, we can use Apache's balancing modules:

```
# Enable the modules
sudo a2enmod proxy_balancer lbmethod_byrequests

# Restart Apache
sudo service apache2 restart
```

Then we can adjust the vhost file:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    <Proxy balancer://mycluster>
        BalancerMember fcgi://127.0.0.1:9000/
        BalancerMember fcgi://127.0.0.1:9001/
        BalancerMember fcgi://127.0.0.1:9002/
    </Proxy>
    <Location />
        ProxyPass balancer://mycluster/
        ProxyPassReverse balancer://mycluster/
    </Location>
    <Location /static>
        ProxyPass !
    </Location>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

This defines a balancer cluster named 'mycluster'. The name can be anything. Then we define our three back-ends. In this case, the back-ends are the three FastCGI listeners.

Then the `<Location...>` directive needs tweaking. We proxy to the cluster named `mycluster` rather than than to the FastCGI process directly. The directive `balancer://mycluster` uses the balancer module to proxy to the cluster.

Using `Location` blocks **won't** currently (as of this writing) work for PHP-FPM due to some bugs. Apache has issues passing the script name when attempting to communicate with PHP-FPM. This stops us from using multiple PHP-FPM back-ends within the same Apache vhost. That's a less common setup in any case.

The `Location` block "style" of proxying requests is best used for applications built in Python, Ruby or others which commonly use FastCGI gateways. These don't have a file-based point of entry like PHP applications do.

Apache with uWSGI

WSGI is a gateway interface originally defined by Python's PEP 333 and updated in PEP 3333[38]. Not surprisingly, this protocol is popular for serving Python applications.

A common implementation of the WSGI protocol is the uWSGI tool. In fact, uWSGI can handle HTTP and FastCGI as well as WSGI, but it's popularly used as a WSGI gateway.

Apache has long worked with uWSGI via a `uwsgi` module available. The `uwsgi` module can work great, however it is complex to configure. A newer and simpler way to use Apache with uWSGI is the `proxy_uwsgi` module. As this makes use of Apache's `proxy` module, the configuration for this will be extremely familiar to you at this point.

There is a "plain" `wsgi` module, but it has been dormant for a while. Supposedly it will be developed on again this year (2014). However uWSGI is a solid, well-developed and current tool. It is what we'll be using here.

A command to use uWSGI with a Flask application found at `myapp/__init__.py` might look something like this: `uwsgi --socket 127.0.0.1:9000 --module myapp --callable=app --stats 127.0.0.1:9191 --master --processes 4 --threads 2`

Let's pretend we have a Python application and are using uWSGI to listen on TCP socket 127.0.0.1:9000. We'll use Apache to proxy to this uWSGI socket using the WSGI gateway protocol.

How to setup a Python application with uWSGI will be the subject of another section or case study of this book. For now, we'll concentrate on Apache's configuration.

First we need to install Apache's dependencies. These includes `proxy_uwsgi`, a module that may not come with Apache out of the box.

[38] http://legacy.python.org/dev/peps/pep-3333/

```
# Install proxy_uwsgi:
sudo apt-get install -y libapache2-mod-proxy-uwsgi

# Enable required modules
sudo a2enmod proxy proxy_uwsgi

# Restart Apache
sudo service apache2 restart
```

Then we can configure an Apache vhost:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    <Proxy *>
        Require all granted
    </Proxy>
    <Location />
        ProxyPass uwsgi://127.0.0.1:9000/
        ProxyPassReverse uwsgi://127.0.0.1:9000/
    </Location>
    <Location /static>
        ProxyPass !
    </Location>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined
```

```
32
33 </VirtualHost>
```

By now this should all be very familiar to you.

The `<Proxy *>` blocks allows access to the proxy from all hosts (all web visitors to our site).

The `<Location />` block is used to accept all requests and proxy them to our uWSGI process listening at 127.0.0.1:9000.

If uWSGI listens on a Unix socket instead of a TCP socket, this would look the following:

```
1 <Location />
2     ProxyPass unix:/path/to/uwsgi.sock|uwsgi:
3     ProxyPassReverse unix:/path/to/uwsgi.sock|uwsgi:
4 </Location>
```

Finally the `<Location /static>` block is what we'll use to serve static content. This configuration assumes there's a directory named `/static`. Any URI which starts with `/static` will be served directly rather than proxied. The `ProxyPass !` directive tells Apache not to proxy the request.

Multiple back-ends

As you may have suspected, we can possibly have multiple uWSGI back-ends as well.

First we need to ensure our needed modules are enabled:

```
1 # Enable the modules
2 sudo a2enmod proxy_balancer lbmethod_byrequests
3
4 # Restart Apache
5 sudo service apache2 restart
```

Then we can adjust the vhost file:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    ServerAlias example.*.xip.io

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted
    </Directory>

    <Proxy balancer://mycluster>
        BalancerMember uwsgi://127.0.0.1:9000/
        BalancerMember uwsgi://127.0.0.1:9001/
        BalancerMember uwsgi://127.0.0.1:9002/
    </Proxy>
    <Location />
        ProxyPass balancer://mycluster/
        ProxyPassReverse balancer://mycluster/
    </Location>
    <Location /static>
        ProxyPass !
    </Location>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/example.com-access.log combined

</VirtualHost>
```

This defines a balancer cluster named ‘mycluster’. The name can be anything. Then we define our three back-ends. In this case, the back-ends are the three uWSGI listeners.

Then the `<Location...>` directive needs tweaking. We proxy to the cluster named `mycluster` rather than than to the uWSGI process directly. The directive `balancer://mycluster` uses the balancer module to proxy to the cluster.

Most often, uWSGI will create separate processes and threads for your Python application. This

means you're more likely to use Apache to proxy requests off to one back-end instead of multiple. Then uWSGI will managing balancing traffic between multiple processes.

However, there are cases where you'll want to run multiple application instances yourself. This might be useful if you use a gateway interface which may not manage multiple processes, such as WSGI with Python's Tornado.

MPM Configuration

At this point, we know enough Apache to get by. However, we have some additional configurations to consider.

Apache has a module system that can determine how requests are processed. These are called Multi-Processing Modules (MPM). They are responsible for communicating over a network, accepting requests, and dispatching child processes/threads to handle requests.

There are three MPM modules to consider:

- MPM Prefork
- MPM Worker
- MPM Event

Before diving in, let's cover some vocabulary.

Processes are "instances" of an application being run. They are isolated and separate from other processes.

A **thread** is something created and owned by a process. Like a process, a thread can execute code. However, a process can have multiple threads. Threads are not isolated from each other - they share some state and memory. This is why you may have heard the term "thread safe". Programs might modify state in one thread, causing errors in code running in another thread.

Threads are smaller than processes as they aren't whole instances of an application. Threads take up less memory, allowing for more concurrent requests. They can also be created and destroyed more quickly than a process. Overall, they are an efficient way to handle web requests, if the code used to handle them is thread safe.

When Apache is started, a master process is created. This process can create more processes. In some instances, those processes spawn threads. While Apache's master process is run as root, processes and threads are created as the configured user and group. These users and groups are usually `www-data` or `apache`.

With that in mind, let's talk about how the three MPM modules use processes and threads to handle requests.

MPM Prefork

MPM Prefork is usually the default MPM used in Apache. It does *not* use threads. An entire process is dedicated to each HTTP request.

> The default may change depending on how you install Apache. For example, the repository we use, `ppa:ondrej/apache2`, enables Event as the default MPM.
>
> If you install the php5 module, this is automatically changed to MPM Prefork. You can see which is installed on Debian/Ubuntu servers at `/etc/apache/mods-enabled/` and see which `mpm_*.conf` file is present.

Because each process handles only one request, Prefork is slightly quicker than a threaded module. There's no processing time spent creating and tracking threads.

While using processes is a little faster, they can eat up CPU and memory in a situation where there is lots of simultaneous requests. A threaded module will be able to handle more concurrent requests.

MPM Worker

MPM Worker uses threading. Each process can spawn multiple threads. Threads are much cheaper to create than processes, and so fewer expensive processes need to be created and managed. This helps Apache handle more concurrent requests by reducing the overall memory needed to handle each request.

With MPM Worker, the processes spawn threads to handle incoming HTTP requests. To be precise, Worker uses one thread per HTTP *connection*. Multiple HTTP requests can be made per connection.

A thread will handle multiple requests until a connection is closed. As a request is completed and a connection closed, the thread opens up to accept the next connection and handle its requests.

MPM Event

MPM Event is the newest processing module. It works just like Worker, except it dedicates a thread to each HTTP request. A thread is created per *HTTP request*, rather than per *connection*.

This means that a thread will free up when the HTTP request is complete, rather than when the connection is closed. Connections are managed within the parent process rather than the threads.

MPM Event is better for applications with relatively long-lasting requests (long Keep-Alive timeouts). With MPM Worker, each long-running connection would use a whole thread. With Event, threads don't need to be taken up by connections which may or may not be sending any data at the moment. A process can use a thread only when a new request comes from the connection.

An application using server-push, long-polling[39] or web sockets are good use cases for employing MPM Event.

[39] http://en.wikipedia.org/wiki/Push_technology

If a connection is made using SSL or TLS, MPM Event defaults back to working just like MPM Worker. It will handle a connection per thread.

MPM Event is stable as of Apache 2.4.

Apache + PHP-FPM Revisited

In Apache, **prefork is always used with `mod_php`** as the PHP5 module is *not* thread safe.

There have been efforts to make PHP common thread safe, so you can use other MPM's that use threads. However it's not "proven" yet. If you're curious, check out building PHP yourself and including php-zts/pthreads.

PHP-FPM gets around the issue of thread-safety by running separately from Apache. This means that we can safely ditch Apache's default MPM Prefork for handling requests!

Let's see an example using MPM Worker.

First, we can install the other MPM modules:

```
# Install both MPMs
sudo apt-get install -y apache2-mpm-event apache2-mpm-worker
sudo service apache2 restart
```

Then we can enable the one we want to use. First, we'll disable the default MPM Prefork, as well as mod_php5, in case it was previously installed/enabled. Then we can enable MPM Worker:

```
# Disable MPM Prefork and php5 so we can enable MPM Event
sudo a2dismod php5 mpm_prefork

# Enable MPM Event
sudo a2enmod mpm_worker

# Restart Apache to load modules
sudo service apache2 restart
```

Now Apache will be using MPM Worker and get all the benefits!

Security Configuration

In Debian/Ubuntu, there's a configuration file for Apache security. This is worth knowing about if you find yourself managing a production system. It's located at `/etc/apache2/conf-available/security.conf`.

The security features are fairly simple. They allow you to control what information is displayed about the server.

This information is visible within the response headers of any HTTP request:

```
$ curl -I localhost
Date: Thu, 26 Jun 2014 19:25:01 GMT
Server: Apache/2.4.7 (Ubuntu)
... other headers...
```

Let's cover some of the options:

ServerTokens

By default, the `ServerTokens` directive is likely set to `OS`. This shows that the web server version and operating system used. On Ubuntu, that might look like "Apache/2.4.7 (Ubuntu)".

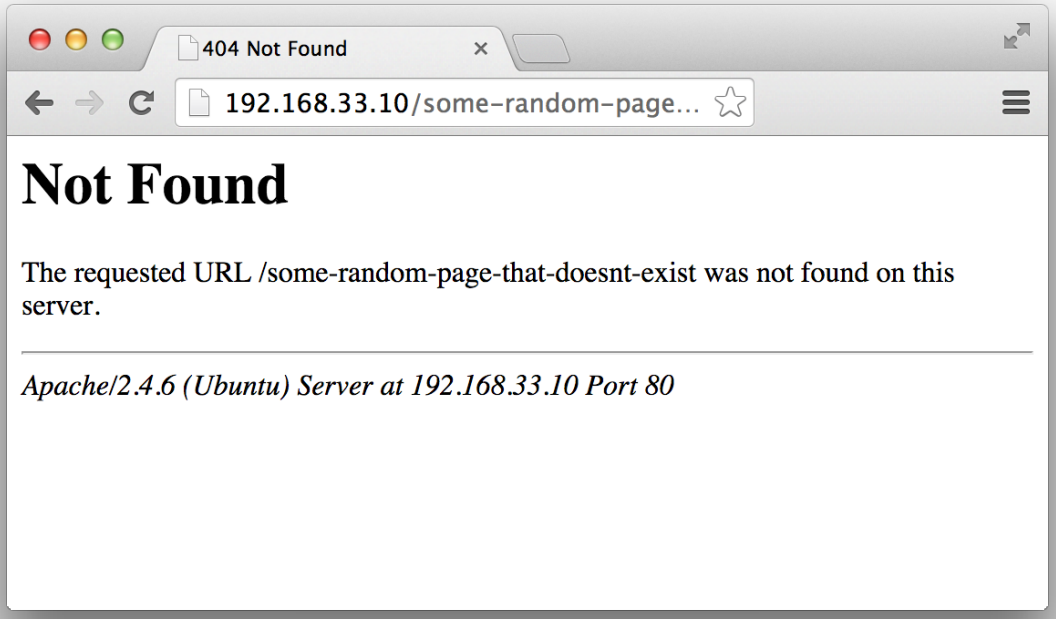
I always set this to `Prod`, which shows the least amount of information - simply that the web server is "Apache".

The Apache documentation says not to worry about showing this information. The general theory is that "Security through obscurity" is not real protection. However, I like hiding the exact Apache version and the Linux distribution used. A specific version may have a security issue. While hiding the version doesn't mean an attacker won't try anyway, you never know!

ServerSignature

The `ServerSignature` directive does what `ServerTokens` does. However this displays server information on generated error pages, such as 404 pages. We can turn this to `Off` to hide the server information.

Here it what the server information looks like on an error page. When we turn server signatures off, thats last line of information is no longer displayed.



Other

There are some other directives in here which I won't cover, but they are worth looking if only to explore your options.

One useful one is an example of preventing access to your version control directories:

```
<DirectoryMatch "/\.svn">
   Require all denied
</DirectoryMatch>
```

You can change that ".svn" to ".git" if you need. That rule will be applied server-wide.

Envvars

In Debian/Ubuntu, there's a configuration file for the environment variables. These are read when Apache is started. This is the `envvars` file, located at `/etc/apache2/envvars`.

We'll cover some of the important configurations found within this file.

APACHE_RUN_USER and APACHE_RUN_GROUP

These are the user and group Apache processes and threads will run as.

The master Apache process is run as user root. That's why we use sudo when we control Apache. However, Apache's child processes and threads will be run as the user and groups specified here.

This is important to know if you have code which performs operations needing permission, such as writing to files. Apache defaults to running as group `www-data`. Consider making the directories written to by your web applications writable via group permissions. This saves you from having to make directories or files 'world-writable'.

For the most part, this is only applicable to PHP run with `mod_php`. Most applications do not run code loaded within Apache. Web applications usually run as a separate process which will have its own user and group settings.

PHP-FPM is one example of such an application. While it is PHP, it's not using Apache's `mod_php`. PHP-FPM has its own user/group configuration. If your PHP application is run using PHP-FPM, then the FPM processes will need permission to write to files/directories. Apache won't need these permissions.

It's worth noting that many web application gateways run as user/group `www-data`. This means your Apache user/group may be the same as your PHP-FPM user/group!

APACHE_LOG_DIR

I don't suggest changing this, but it's good to know that you can change the Apache log directory, and have a place to check to see where it is set.

An example of when you may wish to change this path is if running Apache in a Docker container. This depends on how you choose to handle log files generated processes in Docker containers.

There are other directives in the `envvars` configuration. Adjusting them is less commonly needed. One is performance related: `APACHE_ULIMIT_MAX_FILES`. You may want to increase this if fine-tuning Apache performance. Each process is treated like an open file, so you may max-out the Apache or operating system limit on the maximum number of open files.

MPM Configuration

The `envvars` configuration also has settings to fine-tune the Multi Processing Modules (MPM). We can define how MPM manages processes and threads.

Everything discussed here is also found in Apache's MPM documentation[40].

In some packages/newer versions of Apache, MPM configuration may be found in `/etc/apache2/mods-available/mpm_*.conf` rather than the `envvars` file. You can tell which MPM is enabled by checking to see which `mpm_*.conf` file is present in the `mods-enabled` directory.

Many of these settings are related to how Apache will handle spawning processes and threads. These are worth looking at when attempting to get more performance out of your server.

Remember that the default MPM Prefork only creates child processes. Other MPM's (Worker and Event) spawn processes which create threads.

Here are worthwhile directives to know about. Keep in mind the difference between processes and threads!

MaxConnectionsPerChild - Limits the number of HTTP connections per child process. When the limit is reached, the process dies, to be replaced with a new one. This works regardless of if threading is used (MPM Event/Worker) or not.

MaxRequestWorkers - This limits the number of simultaneous requests being served. Any new request received after this limit is reached is put in a queue.

For MPM Prefork, this means the max number of **processes** launched to handle requests. The default is 256. To increase this, you must also increase the `ServerLimit` directive.

For MPM Worker/Event, this means the max number of **threads** available to serve requests. The default is 400, reached by the following calculation:

16 processes (default `ServerLimit`) * 25 (default `ThreadsPerChild`) = 400 requests.

To increase this passed 16 processes and 25 threads per process, you may need to raise the `ServerLimit` and `ThreadsPerChild` directives.

MaxSpareThreads - The maximum number of idle threads.

Prefork has no threads, so this relates only to threaded MPM's

For MPM Worker/Event, the default is 250. The setting is server-wide, meaning both idle threads and their processes will be killed to reach this number.

Each process/thread takes up a small about of memory, so having a maximum number of idle threads can help save memory usage.

However, idle threads can handle requests more quickly as they don't need to be created before handling a request. Having a fair number of idle workers can help performance, especially for handling request spikes.

[40] http://httpd.apache.org/docs/2.4/mod/mpm_common.html

MinSpareThreads - This is the minimum number of idle threads that can exist. Naturally the minimum shouldn't be higher than the maximum.

Prefork has no threads, so this relates only to threaded MPM's.

For MPM Worker/Event, the default is 75. The setting is server-wide, meaning both idle threads and processes are created until the number is met.

ServerLimit - This is an overall setting which limits other settings we've discussed previously. This is the upper limit on configurable number of processes. All other configurations cannot create more processes than this setting allows.

For MPM Prefork, this simply limits the number of processes set by `MaxRequestWorkers`. Set `ServerLimit` higher if you need to se `MaxRequestWorkers` above the default of 250.

For MPM Worker/Event, this works in conjunction with `ThreadLimit` to set the maximum value for `MaxRequestWorkers`.

Increase `ServerLimit` if you wish to increase the number of processes available. Remember that each process will create new threads until it reaches the `ThreadLimit`. Under MPM Worker/Event, the default is 16 processes.

Apache will try to allocate memory to meet *possible* values of `ServerLimit`, so it should not be set too high. Otherwise memory will be allocated but not used - a waste of resources.

StartServers - This is the number of child processes created at startup. Idle processes and threads allow Apache to quickly respond to new requests. More processes are created as needed on the fly, so this setting may only need adjusting in special cases.

MPM Prefork defaults to 6, while MPM Worker defaults to 3.

StartThreads - The number of idle threads to create on startup. Only relates to threaded MPMs (Worker/Event). Like processes, threads are also created dynamically as needed.

ThreadLimit - Similar to `ServerLimit`, this sets an overall maximum for the server. In this case, it's limiting the number of threads per process, rather than the total number of processes.

The default is 64 threads. Be careful not to set this too much higher than `ThreadsPerChild` due to its potential in wasting unused allocated memory.

ThreadsPerChild - This is the number of threads created by each process. The process creates these threads at startup and never creates more.

The default value is 25 threads. Multiply this by the number of processes in existence to find your total number of threads.

The preceding configurations can all be tweaked to match what your server can handle. The number of configured processes and threads should depend on the CPU cores and RAM available for Apache.

Nginx

> "Apache is like Microsoft Word, it has a million options but you only need six. Nginx does those six things, and it does five of them 50 times faster than Apache." - Chris Lea

Nginx is a lightweight alternative to Apache. Actually, by some metrics, it has overtaken Apache in popularity. Calling it an "alternative" is doing it a disservice.

Nginx is similar to NodeJS, HAProxy, and other "web scale" technologies (put in quotes, only a tad sarcastisically). Nginx runs as an evented, single process. It manages requests asynchronously. This helps Nginx work with a large number of concurrent connections while using a stable and relatively low amount of memory.

Actually Nginx typically uses a few processes. A typical setup with Nginx will spawn as many processes as there are CPU cores on the server.

Apache, as we learned, spawns processes or threads for each connection. Its synchronous manner means that processes and threads pause ("block") while performing slower tasks.

Examples of such tasks are reading from the file system or performing network operations. This means that Apache processes are "blocking"; We must wait for them to finish their task before moving onto the next one.

While Apache spawns many processes and threads, Nginx spawns very few processes ("workers"). Each process is single-threaded. Nginx workers accept requests from a shared socket and execute them inside of an efficient run-loop. Nginx is asynchronous, evented and non-blocking. It is free to accomplish other tasks while waiting for slow tasks such as file I/O or network operations to finish.

Each Nginx worker can process thousands of simultaneous connections. It avoids the overhead of constantly creating, tracking and destroying new processes/threads. This is much more memory and CPU efficient.

Features

Nginx has grown an impressive feature set, most of which is pretty easy to use and setup. Nginx can act as a:

- Web Server
- Reverse Proxy ("Application Proxy")

- Content Caching ("Web Cache")
- Load Balancer
- SSL Terminator

Nginx also has a commercial (paid) version. Notable Nginx Plus features include:

- Advanced load balancing, including dynamically adjusting available servers/nodes
- Advanced caching
- Streaming media abilities
- Monitoring capabilities

Just like in the Apache chapter, we'll cover over the **Web Server** and **Reverse Proxy** functionality.

Installation

We'll use Nginx's "stable" repository for installation. It allows us to get the latest stable versions which can include bug fixes and security updates.

If you have Apache installed on the same server, you'll run into issues starting Nginx, as they both attempt to bind to port 80. You'll need to stop Apache with `sudo service apache2 stop`. I recommend, however, creating a new server if you're following along here on a local virtual machine.

Here's how to install Nginx:

```
sudo add-apt-repository -y ppa:nginx/stable
sudo apt-get update
sudo apt-get install -y nginx
sudo service nginx start

# Set Nginx to start on boot.
# Likely is already set.
sudo update-rc.d nginx defaults
```

Now we can see if this is indeed installed on our server. Let's see if we get an HTTP response:

```
$ curl -I localhost
HTTP/1.1 200 OK
Server: nginx/1.6.1
Date: Thu, 03 Jul 2014 00:49:14 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Thu, 24 Apr 2014 12:52:24 GMT
Connection: keep-alive
ETag: "53590908-264"
Accept-Ranges: bytes
```

Great! We get a response. Nginx is on and working!

Web Server Configuration

In Ubuntu, Nginx follows the usual scheme for configuration. Let's look at some files and directories in the `/etc/nginx` directory:

- `/etc/nginx/conf.d`
- `/etc/nginx/sites-available`
- `/etc/nginx/sites-enabled`
- `/etc/nginx/nginx.conf`

First we have the `sites-available` and `sites-enabled` directories. These work exactly the same way as Apache. Configured servers (aka vhosts) reside in the `sites-available` directory. Configurations can be enabled by symlinking a file from `sites-available` to the `sites-enabled` directory.

Content of the sites-available directory

```
$ cd /etc/nginx
$ ls -la sites-available
[...] root root 4096 Jul  3 01:25 .
[...] root root 4096 Jul  3 01:34 ..
[...] root root 2593 Apr 24 16:23 default
```

We can see this default configuration is symlinked to the `sites-enabled` directory after installation:

Content of the sites-enabled directory

```
$ ls -la sites-enabled/
[...] root root 4096 Jul  3 01:25 .
[...] root root 4096 Jul  3 01:34 ..
[...] root root   34 Jul  3 01:25 default -> /etc/nginx/sites-available/default
```

Unlike Apache, Ubuntu's package of Nginx doesn't include equivalents to `a2ensite` and `a2dissite`. We need to enable/disable site configurations manually.

Inside of `/etc/nginx`, we also can see the main Nginx configuration file `nginx.conf`. Let's see what's interesting in `/etc/nginx/nginx.conf`:

```
# Selections from nginx.conf

include /etc/nginx/mime.types;

##
# Virtual Host Configs
##

include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;
```

We can see that the `mimes.types` configuration is loaded. This configuration simply helps match file extensions to proper mime types.

We can also see that Nginx will attempt to load any file ending in `.conf` in `/etc/nginx/conf.d`. This is similar, but not exactly, like Apache's `conf-available` and `conf-enabled` directory. Apache uses symlinked between the "available" and "enabled" directories. Nginx does not. Instead, any `.conf` file included in `/etc/nginx/conf.d` will be included and enabled.

The last thing I'll note here are these files:

- `/etc/nginx/fastcgi.conf` (formerly `/etc/nginx/fastcgi_params`)
- `/etc/nginx/proxy_params`
- `/etc/nginx/uwscgi_params`

These files contain configurations for using Nginx's reverse proxy abilities. This includes passing requests to FastCGI, uWSGI, or HTTP listeners.

After any configuration change, you can reload Nginx configuration using `sudo service nginx reload`.

You can restart Nginx using `sudo service nginx restart`.

Finally, you can test configuration changes using `sudo service nginx configtest`. This is useful to run after making configuration changes but before reloading/restarting Nginx.

Servers (virtual hosts)

Like Apache, Nginx has the concept of Virtual Hosts, which we'll just call "servers" in context of Nginx.

Unlike Apache, Nginx doesn't make a distinction between IP versus name based virtual hosts. Instead, everything acts as a named-based virtual host.

The following is the default site configuration that comes with Nginx:

File: /etc/nginx/sites-available/default, with comments stripped out

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    server_name localhost

    charset utf-8;

    location /  {
        try_files $uri $uri/ =404;
    }
}
```

This is a basic server. Let's cover what we're seeing here:

Directive	Explanation
Listen	First we can see that it listens on port 80, and also defines itself as the default server for requests on port 80. If no `Host` HTTP header matches a configured server, then Nginx will default back to this default site. You can define multiple defaults. For example a default_server on port 8080: `listen 8080 default_server` is different from the default site on port 80: `listen 80 default_server` We also listen on port 80 of an ipv6 interface, if it is enabled on the server
root	Here we define the document root. This is where the web files are pulled from. This is equivalent to Apache's `DocumentRoot` directive
index	The index directive defines which files are to be attempted to be read if no file is specified & is equivalent to Apache's `DirectoryIndex` directive.
server_name	The hostname that Nginx should use to match the `Host` header with to match to this server. Since this is a default server (via `default_server`), currently this site will load if no other host is matched. You can use multiple names, such as `server_name www.example.com example.com` You can define wildcards on a server names beginning or end, such as `server_name *.example.com` You can use regex to match more complex needs as well. For example `~^(.*)\.example\.com$` matches any subdomain of `example.com`. We can assign regex capture groups to a variable to be used later in the configuration. You can use regex to match more complex needs as well, such as `~^(.*)\.example\.com$`, which has the benefit of letting us capture and use the matched portion of the regex.
charset	Always use utf-8. If you ever start creating a web application using another character set, you'll be shooting yourself in the foot.
location	Nginx can use the location block along with a file path or regex pattern to match URL's or files and handle them differently. Here we see any location is grabbed. Then the `try_files` directive will attempt to find a file in the order of the given patterns. By default, this tries to use the explicit URL to find a file, followed by a directory name, and lastly responds with a `404` if no matching file or directory is found.

Location Block

The `location` directive is very important. It helps determine how files and URI's are handled by Nginx.

For example, we saw our default block:

```
location / {
    try_files $uri $uri/ =404;
}
```

This captures the URI "/" and any sub-URI (sub-directory). In other words, this location block applies to all URIs.

The use of `try_files` is good for handling static content. It tries to find the URI as a file or directory in the order of the defined variables. The order we see here will tell Nginx to find files in this order:

- First try the URI given to see if a matching file can be found on the server. This is relative to the `root` path.
- Failing to find a file, try the URI as a directory on the server. This is relative to the `root` path.
- Failing to find a file or a directory, respond with a `404` error.

Let's see some other example location blocks.

"Boring" files:

First, we might want to handle favicons and robots.txt files differently. They are frequently missing and often requested by browsers and site crawlers. These can eat up our server logs with unimportant `404` errors.

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

The above two directives will turn off `404` error logging and any access log information on these two files.

Blocking Access to Files

Next, let's see how to block access to some files. Normally we don't want to serve files or directories beginning with a period. These include `.git`, `.htaccess`, `svn` and others:

```
location ~ /\. {
    deny all;
    access_log off;
    log_not_found off;
}
```

This turns off the access log and `404` error logging for "dot files". If the files or directories exist on the server, Nginx will deny access to them.

Handling Files by Extension

Next, let's see how to handle files with specific extensions:

```
location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    try_files $uri =404;
}
```

This uses regular expressions to match files `.js`, `.css` and the others listed above. The above uses `try_files` for the exact file name but doesn't attempt to match them as a directory.

This is useful for setting cache expiration headers for static assets. See H5BP's Nginx configuration repository[41] for examples.

Matching by extension in this manner is similar to how we will handle PHP file requests in an upcoming section:

```
location ~ \.php {
    ...magic here...
}
```

As we'll see later, this will be used to set any file ending in `.php` to being processed by the PHP interpreter. We can reduce the risk of unwanted PHP files being processed by explicitly specifying which PHP files can be run:

```
location ~ ^/(app|app_dev|config)\.php(/|$) {
    ...magic here...
}
```

This only allows `app.php`, `app_dev.php`, and `config.php` files. These happen to be used by the Symfony PHP framework.

Pretty URL's (Hiding 'index.php')

PHP developers often want to hide the `index.php` file from the URL. In Apache, this is done with the Rewrite module. With Nginx, we can use the `try_files` directive:

[41] https://github.com/h5bp/nginx-server-configuration

```
# With a PHP application, this
# becomes our default `location {}` block
location / {
    try_files $uri $uri/ /index.php$is_args$args;
}
```

This will use `try_files` to find a file or directory based on the URI given. Finally, it will try to use `index.php` and pass it the arguments. In this way, all not-found files or directories will fall back to our application's `index.php` file. Nginx won't respond with a `404` error directly.

The `location` block handling all PHP files will then pick up from there and handle the request.

Redirects and Other Tricks

Similar to Apache, Nginx does have a `rewrite` ability. However the use of RegEx is generally considered inefficient[42]. Instead, we can often use the `server` directive along with `server_name` and redirect as needed. Let's see some examples of that.

Redirect www to non-www

While there are some good reasons to use the `www` subdomain[43], there may be times when you wish to force the use of the root domain. To accomplish that, we'll add in a server block to detect a non-www url being used, and redirect to a www version of it:

```
server {
    server_name  *.example.com;
    return       301 $scheme://example.com$request_uri;
}

server {
    server_name example.com;
    ...
}
```

The above will capture *any* subdomain of `example.com` and redirect it to the root domain.

If your server handles requests for other subdomains, you may instead wish to only redirect the `www` subdomain to the non-`www` subdomain for your main site:

[42] http://wiki.nginx.org/Pitfalls#Taxing_Rewrites
[43] http://www.yes-www.org/why-use-www/

```
server {
    listen       80;
    server_name  www.example.com;
    return       301 $scheme://example.com$request_uri;
}

server {
    listen 80;
    server_name  example.com;
    ...
}
```

The top `server` block listens for requests made to `www.example.com` and redirects to the non-www version of the URL.

Redirect non-www to www

If you fall into the "yes-www" camp, you can do the inverse to ensure the "www" is used:

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

server {
    listen       80;
    server_name  www.example.com;
    ...
}
```

Forcing SSL

If you need your site URLs to use HTTPS, you can use a similar technique. The following listens on port 80 for any "http" requests and redirects them to the its "https" version.

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

server {
    listen       443 ssl;
    server_name  example.com;

    ssl on;
    # Other SSL directives, covered later
    ...
}
```

The above also redirects to the non-www domain. Which you redirect to is up to you.

Wildcard Subdomains and Document Root

For development, it might be useful to have a setup where each directory you create in a folder maps to a separate website.

Imagine if a url `project-a.local.dev` mapped to document root `~/Sites/project-a/public`. Then, a url `project-b.local.dev` mapped to document root `~/Sites/project-b/public`. That might be really useful if you didn't want to change server settings for each of the sites you worked on!

Above, we noted that `server_name` can take wildcards, and regular expressions. We'll make use of regular expressions to map a subdomain to a document root. Let's see what that looks like:

```
server {
    listen 80 default_server;

    server_name ~^(.*)\.local\.dev$;
    set $file_path $1;

    root /var/www/$file_path/public

    location / { ... }
}
```

We're using regular expressions in the `server_name` directive. This matches any subdomain and captures the subdomain. The subdomain is available via the `$1` variable. The `$1` variable is the result of the first capture group found in the regular expression `~^(.*)\.local\.dev$`.

We then use the `$1` variable and map it to a variable called `$file_path`. Lastly, we append `$file_path` to the `root` directive to make up part of our document root. This will dynamically change the document root based on the subdomain used.

Each subdomain will automatically map to our project directories!

Note that I assume the domain `local.dev` and any of its subdomains will point to your web server. This might not be the case unless you edit your computer's `hosts` file.

Integration with Web Applications

Nginx wouldn't be nearly so useful if we couldn't use it to send requests to our web applications.

Typically a web server will accept a request and pass it off to a "gateway". Gateways then translate and pass the request off to a coded application. Gateways are various implementations and flavors of a "CGI"s - a Common Gateway Interfaces[44].

For Python applications, communication is often accomplished with a WSGI[45] gateway. Nginx sends requests off to a WSGI gateway, which in turns passes a request to the Python application.

For PHP, this means Nginx sends a request off to PHP-FPM. PHP-FPM is a FastCGI[46] gateway. Nginx will convert request information to FastCGI. PHP-FPM accepts that FastCGI request and sends it to our application.

Nginx can also proxy requests to web applications over HTTP. This is popular when sending requests to applications directly, without a gateway. NodeJS or Golang are two languages which can natively "speak" HTTP.

Some gateways prefer to speak HTTP as well. Unicorn and Gunicorn are two gateways which accept HTTP requests before sending them off to Ruby or Python applications.

Here we'll discuss how Nginx can talk to applications using HTTP, FastCGI and WSGI gateways.

Nginx isn't limisted to those three protocols, however. It can act as a reverse proxy for the following protocols:

- **HTTP** - Other web servers or perhaps NodeJS, Go apps or HTTP gateways such as Unicorn/-Gunicorn (Ruby, Python)
- **FastCGI** - Many application gateways can also FastCGI from Unicorn/Gunicorn to PHP-FPM (Ruby, Python, PHP)
- **uWSGI** - Primarily used for Python applications with uWSGI

[44] http://en.wikipedia.org/wiki/Common_Gateway_Interface
[45] http://wsgi.readthedocs.org/en/latest/
[46] http://www.fastcgi.com/drupal/

- **SCGI** - Another CGI implementation
- **Memcached** - Proxying requests to Memcached

Nginx is a "reverse" proxy because it dispatches a single request off to (potentially) multiple services. A regular (forward) proxy does the inverse. A load balancer is another example of a reverse proxy.

Sockets

It's worth taking a second for a reminder that Nginx, like Apache, can proxy to both TCP and Unix sockets. What is a socket? A socket itself is just a "thing" a process can use to send and receive data. It's a connection used for communication. There are two main kinds of sockets:

A **TCP socket** is the combination of an IP address and a port number. HTTP uses tcp sockets to make HTTP requests. TCP sockets work over your servers network, and can reach across networks to remote servers.

A **Unix socket** is a pseudo file which acts as a socket. These work a bit faster then TCP sockets, but are limited to the local filesystem of a server. Because they work on the filesystem, you can use the usual permissions to control access to them.

HTTP Proxy

We'll start by seeing what Nginx can do when passing a request off to a process which happens to also be listening over HTTP. Let's pretend any request sent to the `/api` route (or any subdirectories of it) should go to an application listening on localhost port 9000:

```
location /api {
    include proxy_params;
    proxy_pass http://127.0.0.1:9000;
}
```

What did we do here?

We included the `/etc/nginx/proxy_parms` file. This file contains some sensible defaults to use when proxying requests for another service. Here's what that file does:

- It sets the `Host` header to the requests original `Host`
- It adds a `X-Read-IP` header to the IP address of the original request
- It adds a `X-Forwarded-For` header
- It adds a `X-Forwared-Proto` header

These headers are all commonly used for web applications behind a load balancer or other reverse proxy. A web application can use these to know the information about the origin request. If these directives were not available, every request would look like it came from Nginx!

Nginx then proxies the request off to the server via the `proxy_pass` directive. Nginx will return to the client whatever the backend server returns.

A Unix socket version of the same proxy pass might look like this:

```
location /api {
    include proxy_params;
    proxy_pass unix:/path/to/socketfile.sock;
}
```

Here's a more complete virtual host configuration:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    server_name localhost

    charset utf-8;

    location /  {
        try_files $uri $uri/ =404;
    }

    location /api {
        include proxy_params;
        proxy_pass http://127.0.0.1:9000;
    }
}
```

Multiple Backends

Nginx can proxy off to multiple HTTP backends. In fact, this is Nginx's load balancing!

A quick example of proxying to multiple HTTP backends would look like this. Note that this will be covered in more detail in the Load Balancing chapter.

```
upstream my_app {
    zone backend 64k;
    least_conn; # Discussion on LB algorithms in the LB chapter!
    server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
}

server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    server_name localhost

    charset utf-8;

    location /static {
        try_files $uri $uri/ =404;
    }

    location / {
        include proxy_params;
        proxy_pass http://my_app/;
    }
}
```

We can also see here that I'm telling Nginx to serve static files if they are in the `/static` directory (or a subdirectory of it). All other URLs are passed to the proxy.

FastCGI

Another common way to proxy pass a request to an application gateway is using the FastCGI protocol. This is how we use Nginx to talk to PHP-FPM, which is a FastCGI gateway implementation for PHP.

Nginx can, of course, speak to any FastCGI process. You might find this used with uWSGI, Unicorn or Gunicorn gateway interfaces, all of which can "speak" FastCGI.

Here's a PHP-FPM example. Where earlier we listened for any url or sub-url of the `/api` uri, here we'll listen for any request ending in `.php`:

```
location ~ \.php$ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;

    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;

    include fastcgi.conf; # fastcgi_params for nginx < 1.6.1
    fastcgi_param PATH_INFO       $fastcgi_path_info;
    fastcgi_param ENV development;
}
```

Let's cover these:

- `fastcgi_split_path_info` - Helps get the path after the PHP file in the URI. This is helpful since we are commonly hiding the `index.php` file from the URL in our applications.
  - Given the URL `/subdirectory/index.php/some/uri`, the PATH_INFO will becomes `/some/uri`. Our application likely expects this path for routing purposes.
  - We then can set the `PATH_INFO` variable parameter with the path information created.
- `fastcgi_pass` - Pass the request off to a socket
- `fastcgi_index` - Set the filename to be appended to the end of directory. This is similar to setting the `index` directive for static files in Nginx.
- `include fastcgi.conf` - Similar to `proxy_pass`, Nginx has some sane defaults to pass to any FastCGI process. There are many. I highly suggest you check out the parameters being passed within `/etc/nginx/fastcgi_params`.
- `fastcgi_param` - Pass any arbitrary parameter to the FastCGI process. These will be made available as an environmental variable. With PHP, they are in the `$_ENV` and `$_SERVER` globals.
  - These work in the format `fastcgi_param KEY VALUE`.

A more complete virtual host for PHP-FPM might look like this:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    server_name localhost

    charset utf-8;

```

```
12      location /  {
13          try_files $uri $uri/ /index.php$is_args$args;
14      }
15
16      location ~ \.php$ {
17          fastcgi_split_path_info ^(.+\.php)(/.+)$;
18
19          fastcgi_pass 127.0.0.1:9000;
20          fastcgi_index index.php;
21
22          include fastcgi.conf; # fastcgi_params for nginx < 1.6.1
23          fastcgi_param PATH_INFO       $fastcgi_path_info;
24          fastcgi_param ENV development;
25      }
26  }
```

The `try_files $uri $uri/ /index.php$is_args$args;` portion will pass requests off to PHP last if no directory or static file is found to serve the request.

Not PHP

If we have an application that is not PHP, then we likely don't have a file extension to match a request against. URI's in such applications are almost always directories. However, PHP almost always uses an `index.php` file, even if it's hidden from the URL.

Applications written in pretty much anything that isn't PHP usually base routes on directory URIs. PHP is in fact the outlier in its behavior; It's treated more like a static file that happens to have code in it.

In such a situation, we need a way to pass all requests off to our application unless they are a static file. A typical setup is to reserve a directory to use for static assets. This lets us make Nginx behave as follows:

- Serve any static file directory from the `/static` directory or subdirectories
- Send all other requests to our application

We can do that using two `location` blocks:

```
location /static {
    try_files $uri $uri/ =404;
}

location /  {
    include fastcgi.conf;  # fastcgi_params for nginx < 1.6.1
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_param ENV development;
}
```

This passes a request off to our FastCGI listener if a file or directory from the `/static` directory is not requested. Note that the FastCGI parameters are simplified. We don't need to take the file path before and after a `.php` file into account. We just pass the whole URI and query off to our application via FastCGI.

A consequence of this method is that Nginx handles the `404` response. In our previous setup, we passed that responsibility to the proxied application.

The Nginx Pitfalls wiki[47] page also has an interesting way of handling static vs non-static files. This is more elegant than a reserved "static" directory:

```
location / {
    try_files $uri $uri/ @proxy;
}

location @proxy  {
    include fastcgi.conf;  # fastcgi_params for nginx < 1.6.1
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_param ENV development;
}
```

This attempts to find the URI as an existing file or directory. If they don't exist, it jumps the request to the `@proxy` location block. This will then proxy the request to the back-end server (application) configured.

Multiple Backends

Nginx can proxy off to multiple FastCGI backends.

A quick example of proxying to multiple FastCGI backends would look like this:

[47] http://wiki.nginx.org/Pitfalls#Proxy_Everything

```
upstream my_app {
    zone backend 64k;
    least_conn;
    server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
}

server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    server_name localhost;

    charset utf-8;

    location / {
        try_files $uri $uri/ @proxy;
    }

    location @proxy {
        include fastcgi.conf; # fastcgi_params for nginx < 1.6.1
        fastcgi_pass my_app;
        fastcgi_param ENV development;
    }
}
```

We simply tell Nginx to `fastcgi_pass` to the `my_app` upstream backend.

uWSGI

As covered in the Apache chapter, Python often uses WSGI as a gateway interface for web servers to communicate to Python applications. The uWSGI gateway is a common implementation of the WSGI specification.

Nginx, luckily, can "speak" (u)WSGI natively. Let's take a look at a setup we can use for that:

```
location / {
    try_files $uri $uri/ @proxy;
}

location @proxy  {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:9000;
    uwsgi_param ENV productionmaybe;
}
```

This is exactly like our FastCGI implementation, except we switch out FastCGI for uWSGI!

Note that we also include Nginx's `uwsgi_params`[48] file. This is similar to the FastCGI parameters configuration file. It passes information used by uWSGI and potentially by our applications to fulfill HTTP requests.

Multiple Backends

Nginx can proxy off to multiple uWSGI backends.

A quick example of proxying to multiple uWSGI backends would look like this:

```
upstream my_app {
    zone backend 64k;
    least_conn;
    server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
}

server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    server_name localhost

    charset utf-8;

    location /static {
        try_files $uri $uri/ @proxy;
```

[48] http://uwsgi-docs.readthedocs.org/en/latest/Nginx.html#what-is-the-uwsgi-params-file

```
21      }
22
23      location @proxy {
24          include uwsgi_params;
25          uwsgi_pass my_app;
26          uwsgi_param ENV productionmaybe;
27      }
28  }
```

PHP

PHP is still the most-used language in the web and is therefore well worth discussing. PHP traditionally has been used with Apache by embedding PHP into it. However more modern PHP can be used with PHP-FPM, an implementation of FastCGI.

Let's go over PHP and make some notes about its configuration and the various ways it is used.

Installation

Like much of the software we discuss in this book, there's a good repository available to use. The `ppa:ondrej/php5` repository will allow us to install the latest stable version of PHP:

```
# Add the repository
sudo add-apt-repository -y ppa:ondrej/php5

# Update the repositories
sudo apt-get update
```

Then we can install the latest PHP:

Install PHP and PHP CLI

```
sudo apt-get install -y php5 php5-cli
```

PHP has quite a few modules - there are quite a few common ones to install. Here are the ones I most often install.

Install common PHP modules

```
# A good base-line PHP install

# PHP "common" along with CLI php and other common modules:
sudo apt-get install -y php5 php-cli php5-curl php5-mcrypt php5-intl php5-gmp

# Some database (and cache) specific modules (will also install PDO)
sudo apt-get install -y php5-mysql php5-pgsql php5-sqlite  php5-memcached

# Image processing
```

```
sudo apt-get install -y php5-gd php5-imagick

# Debugging, likely not to be installed in production
sudo apt-get install -y php5-xdebug
```

You can install these all in one shot:

```
sudo apt-get install -qq php5 php5-cli php5-mysql php5-pgsql \
php5-sqlite php5-curl php5-gd php5-gmp php5-mcrypt php5-xdebug \
php5-memcached php5-imagick php5-intl
```

These packages enable the modules once installed, so you shouldn't need to enable them yourself.

Configuration

The configuration files for PHP are located in `/etc/php5`. The following directories are commonly found within the PHP configuration:

- **/etc/php5/apache2** - If Apache is installed, this directory controls PHP configuration for Apache
- **/etc/php5/fpm** - If PHP-FPM is installed, this directory controls PHP configuratoin for PHP-FPM
- **/etc/php5/cli** - Controls PHP configuration for CLI-based PHP
- **/etc/php5/mods-available** - *All* PHP modules available for use

PHP can be configured separately for each context in which it's used. We can have a different `php.ini` configuration and load separate modules when PHP is used with Apache2's `mod_php`, when used with PHP-FPM, and when used on the command line.

This is often a source of confusion. Sometimes we'll see PHP that works fine in context of a web browser suddenly not work when the same code is called on the command line.

This is usually because the configurations are different for command-line PHP than for PHP run in Apache or FPM. For example, if the Mcrypt module could be loaded for use in Apache but not for use in CLI.

Let's take a closer look inside of `/etc/php5/apache2`:

- **php.ini** - The INI file for PHP used within Apache's `mod_php`

- **conf.d** - A directory of symlinks pointing to loaded modules from `/etc/php5/mods-available` for use within Apache. This is how we control what modules are loaded when using PHP with Apache.

As stated, this lets us control how PHP is configured depending on the contex it is used (cli, apache, php-fpm). For PHP, this is both run-time configuration, via the php.ini file, as well as the modules loaded.

For each context, modules are loaded in alpha-numeric order from the `conf.d` directory. Let's see the `conf.d` directory for Apache:

An abbreviated list of files found in Apache's 'conf.d' directory

```
$ cd /etc/php5/apache2/conf.d
$ ls -la
root root Jun 24 01:07 .
root root Jun 24 01:32 ..
root root Jun 24 01:07 05-opcache.ini -> ../../mods-available/opcache.ini
root root Jun 24 01:07 10-pdo.ini -> ../../mods-available/pdo.ini
root root Jun 24 01:07 20-json.ini -> ../../mods-available/json.ini
root root Jun 24 01:07 20-readline.ini -> ../../mods-available/readline.ini
... More unlisted here...
```

The modules listed here are all symlinks (aliases) to modules in the `/etc/php5/mods-available` directory. Modules inside of `/etc/php5/mods-available` are the pool of available modules - we decide which are loaded when PHP is used with Apache by creating the symlinks to files the `/etc/php5/apache2/conf.d` directory.

We can see the modules loaded when Apache uses PHP. Note the file names of the symlinks are preceded with a number so that the order they are loaded can be set.

Helper Commands

The Debian/Ubuntu packages for PHP provide some helper tools to enable and disable PHP modules:

```
# Enable PHP's mcrypt for apache2
sudo php5enmod apache2 mcrypt

# Or disable it:
sudo php5dismod apache2 mcrypt
```

Here we can replace "apache2" with "fpm" or "cli" to affect the desired context. The second argument is the name of any module listed in the `/etc/php5/mods-available/` directory.

PHP-FPM

PHP-FPM provides another popular way to use PHP. Rather than embedding PHP within Apache, PHP-FPM allows us to run PHP as a separate process.

PHP-FPM is a FastCGI implementation for PHP. When the web server detects a PHP script is called, it can hand that request off (proxy it) to PHP-FPM using the FastCGI protocol.

Some benefits of PHP-FPM:

- PHP-FPM runs separately from the web server, reducing memory used for requests that are not PHP-related
- Web servers can do what they do best - simply serve static content
- PHP-FPM can be run on multiple servers, distributing server load
- PHP-FPM can run different "resource pools", allowing for separate configurations per pool

This is how most web applications are run, whether they are coded in PHP, Python, Ruby or other web-languages. The application will run as a separate process, which a web server can proxy requests off to. PHP developers might be more used to Apache and PHP "just working" together without having to think about it.

Apache

When Apache uses `mod_php`, it actually loads PHP on *each* request! By eliminating `mod_php`, we reduce the overall memory used. The result is that web servers can handle more (concurrent) requests!

PHP-FPM isn't necessarily *faster* than Apache's `mod_php`. Instead, FPM's efficient use of memory gives us the ability to handle more traffic per server.

I recommend leaving `mod_php` behind for good. The benefits of ditching `mod_php` in favor of PHP-FPM are too high!

Nginx

Nginx, on the other hand, *requires* the use of PHP-FPM if you want to run a PHP application. It doesn't have a module to load in PHP like Apache can. I always use Nginx, as it does everything I need with a simpler configuration, and better overall performance.

Process Management

PHP-FPM's master process creates child processes to handle all PHP requests. Processes are expensive to create and manage. How we treat them is important.

PHP-FPM is an implementation of FastCGI, which uses "persistent processes". Rather than killing and re-creating a process on each request, FPM will re-use processes.

This is much more efficient than Apache's `mod_php`, which requires Apache to create and destroy a process on every request.

Install PHP-FPM

To install PHP-FPM, we'll use the package "php5-fpm":

```
1 sudo apt-get install -y php5-fpm
```

As we mentioned, PHP-FPM runs outside of Apache, so we have another service we can start, stop, reload and restart:

```
1 sudo service php5-fpm start
```

It's important to note that generally you will always use PHP-FPM in conjunction with a web server "in front" of it. This is because PHP-FPM doesn't handle web requests directly (using HTTP).

Instead, it communicates with the FastCGI protocol. In order to process a web request, we need a web server capable of accepting an HTTP request and handing it off to a FastCGI process.

Configuring PHP-FPM

Configuration for PHP-FPM is all contained within the `/etc/php5/fpm` directory:

```
$ cd /etc/php5/fpm
$ ls -la
drwxr-xr-x 4 root root  4096 Jun 24 15:34 .
drwxr-xr-x 6 root root  4096 Jun 24 15:34 ..
drwxr-xr-x 2 root root  4096 Jun 24 15:34 conf.d
-rw-r--r-- 1 root root  4555 Apr  9 17:26 php-fpm.conf
-rw-r--r-- 1 root root 69891 Apr  9 17:25 php.ini
drwxr-xr-x 2 root root  4096 Jun 24 15:34 pool.d
```

As you can see, the FPM configuration includes the usual `php.ini` file and `conf.d` directory. FPM also includes a global configuration file `php-fpm.conf` and the `pool.d` directory. The `pool.d` directory contains configurations for FPM's resource pools. The default `www.conf` file defines the default pool.

Here are some information on PHP-FPM configuration:

Global Configuration

The first thing we'll look at is FPM's global configuration, found at `/etc/php5/php-fpm.conf`. Unless making specific performance tweaks, **I leave this file alone**. There's still some interesting information we can gleam from this.

error_log = /var/log/php5-fpm.log We can see the error log for FPM is located at `/var/log/php5-fpm.log`.

log_level = notice The log level of reporting to the error log. By default this is set to `notice`, but can be alert, error, warning, notice or debug. Set these to more verbose logging (debug or notice) and restart FPM for debugging purposes only.

emergency_restart_threshold = 0 This is an integer representing the number of child processes to exit with errors that will trigger a graceful restart of FPM. By default, this is disabled (value of zero).

emergency_restart_interval=0 Interval of time used to determine when a graceful restart will be initiated. By default, this is an integer in seconds, but you can also define minutes, hours and days. This works in conjunction with emergency_restart_threshold.

daemonize = yes Run PHP-FPM as a daemon, in the background. Setting this to 'no' would be a less common use case. Uses for not daemonizing may include:

1. Debugging
2. Use within a Docker container
3. Monitoring FPM with a monitor which prefers processes are not run as a daemon

include=/etc/php5/fpm/pool.d/*.conf Include any configuration files found in `/etc/php5/fpm/pool.d` which end in the `.conf` extension. By default, there is a `www.conf` pool, but we can create more if needed. More on that next.

Resource Pools

Here's where PHP-FPM configuration gets a little more interesting. We can define separate resource "pools" for PHP-FPM. Each pool represents an "instance" of PHP-FPM, which we can use to send PHP requests to.

Each resource pool is configured separately. This has several advantages.

1. Each resource pool will listen on its own socket. They do not share memory space, a boon for security.
2. Each resource pool can run as a different user and group. This allows for security between files associated with each resource pool.
3. Each resource pool can have different styles of process management, allowing us to give more or less power to each pool.

The default `www` pool is typically all that is needed. However, you might create extra pools to run PHP application as a different Linux user. This is useful in shared hosting environments.

If you want to make a new pool, you can add a new `.conf` file to the `/etc/php5/fpm/pool.d` directory. It will get included automatically when you restart PHP-FPM.

Let's go over some of the interesting configurations in a pool file. You'll see the following in the default `www.conf` file. In addition to tweaking the www pool, you can create new pools by copying the `www.conf` file and adjusting it as needed.

Pool name: www At the **top** of the config file, we define the name of the pool in brackets: [www]. This one is named "www". The pool name needs to be unique per pool defined.

Conveniently, the pool name is set to the variable `$pool`. This can be used anywhere within the configuration file after it is defined.

user=www-data & group=www-data If they don't already exist, the php5-fpm package will create a `www-data` user and group. This user and group is assigned as the run-as user/group for PHP-FPM's processes.

It's worth noting that PHP-FPM runs as user root. However, when it receive a new request to parse some PHP, it spawns child processes which run as this set user and group.

This is important in terms of Linux user and group permissions. This `www-data` user and group lets you use Linux permissions to lock down what the PHP process can do to your server.

This setting is one of the reasons why you might create a new resource pool. In a multi-site environment, or perhaps in a shared hosting environment, you can create a new pool per user. So if each Linux user (say Chris, Bob and Joe all are users on this server) wants to run their own sites, a new pool can be created for each user. Their pools won't interact with each other as they are configured separately. This will ensure that PHP run as user Bob won't be able to read, write to or execute files owned by Joe.

The `user` and `group` setting should always be set to an already existing server user/group. You can read more on user and group permissions in the Permissions and User Management chapter.

listen = /var/run/php5-fpm.sock By default, PHP-FPM listens on a Unix socket.

A "socket" is merely a means of communication. Unix sockets are faux-files which work to pass data back and forth. A TCP socket is the combination of an IP address and a port, used for the same purpose.

A Unix socket is a little faster than a TCP socket, but it is limited in use to the local file system.

If you know your PHP-FPM process will always live on the same server as your web server, then you can leave it as a Unix socket. If you need to communicate to PHP-FPM on a remote server, then you'll need to use the network by using a TCP socket.

Changing this to a TCP socket might look like this:

```
1 listen = 127.0.0.1:9000
```

This listens on the loopback network interface (localhost) on port 9000. If you need to enable PHP-FPM to listen for remote connections you will need to bind this to other network interfaces:

```
1 # Binding to network 192.168.12.*
2 listen = 192.168.12.12:9000
```

You can have PHP-FPM listen on all networks. This is the least secure, as it may end up listening on a publicly-accessible network:

```
1 # If you are binding to network 192.168.12.*
2 listen = 0.0.0.0:9000
```

For each resource pool created, the `listen` directive needs to be set to a unique socket.

listen.owner / listen.group & listen.mode If you use a Unix socket instead of a TCP socket, then you need to set the user/group permissions of the socket file.

Since Unix sockets are faux-files, they (in most cases) follow the same permission rules of Linux files. The socket file usually needs to have read/write permissions open to the file owner.

Using a Unix socket should "just work" by default. The user/group is set to `www-data` by default, with its permissions set to `0600`. Only the file owner can read and write to it.

If you change the user/group setting of a resource pool, you should also change this to the same user/group.

listen.allowed_clients = 127.0.0.1 If you are using a TCP socket, then this setting is good for security. It will only allow connections from the listed addresses. By default, this is set to "all", but you should lock this down as appropriate.

This only applies to TCP sockets as Unix sockets can only be used locally and are not related to the network.

A good set of firewall rules will block external connections to the PHP-FPM processes. This provides some redundancy in limiting who can connect to the FPM processes.

You can define multiple addresses. If you need your loopback (127.0.0.1) network AND another server to connect, you can do both:

```
1  # Multiple addresses are comma-separated
2  listen.allowed_clients = 127.0.0.1, 192.168.12.12
```

This setting pairs with the **listen** directive described above. If you listen on any network interface other than the loopback (localhost, 127.0.0.1), you should also adjust this directive.

Currently, only ipv4 addresses can be defined. You cannot use hostnames or ipv6 addresses.

pm = dynamic Process management is set to **dynamic** by default. The dynamic setting will start FPM with at least 1 child process waiting to accept a connection. It will dynamically decide how many child processes are active or waiting on a request. This uses other settings we'll discuss next to manage processes.

The `pm` directive can also be set to **static**. This sets a specific number of child processes. This number of processes is alway present regardless of other settings.

Lastly, the `pm` directive can be set to **ondemand**. This is the same as **dynamic**, except there's no minimum number of child processing created.

pm.max_children = 5 The maximum number of child processes to exist at any point. This sets the overall maximum number of simultaneous requests PHP-FPM will handle.

Increasing this will allow for more requests to be processed concurrently. However there are diminishing returns on overall performance due to memory and processor constraints.

Nginx starts with a low number (5), since Ubuntu packages tend to optimize for low-powered servers. A rule of thumb for figuring out how many to use is:

```
pm.max_children = (total RAM - RAM used by other process) / (average amount of R\
AM used by a PHP process)
```

For example, if:

- The server has 1GB of ram (1024mb)
- The server has an average baseline of 500mb of memory used
- Each PHP process takes 18mb of memory

Then our `max_children` can be set to 29, much higher than the default of 5!

> That math was: ((1024-500)/18 = 29.111). I rounded down to be conservative.

You'll need some investigation to figure these numbers out. Pay special attention to what else you run on your server (Nginx, MySQL and other software).

Using a database or cache on the same server especially makes this a tricky calculation. Memory usage can spike, resulting in PHP-FPM competing for resources. This will likely cause the server to start "swapping" (using hard drive space as overflow for RAM), which can slow a server down to a halt.

If you have more than one resource pool defined, you need to take process management settings into account. Each pool has a separate set of processes that will compete for resources.

In any case, on a server with 1GB of RAM, your number of `max_children` should be higher than the default 5. However, this depends on what else is installed.

pm.start_servers = 2 The number of processes created by PHP-FPM on startup. Because processes are expensive to create, having some created at startup will get requests handled more quickly. This is especially useful for reducing startup time on busy servers. This only applies when process management is set to "dynamic".

pm.min_spare_servers = 1 The *minimum* number of processes PHP-FPM will keep when there are no requests to process (when idle). Because processes are expensive to create, having some "idle" will get requests processed quickly after a period of idleness.

pm.max_spare_servers = 3 This is the number of "desired" spare servers. PHP-FPM will attempt to have this many idle processes ready, but will not go over the maximum set by `pm.max_children`. If `pm.max_children` is set to `5`, and there are `4` processes in use, then only one spare (idle) process will be created. This only applies when process management is set to "dynamic".

pm.process_idle_timeout = 10s The number of seconds a process will remain idle before being killed. This only applies when process management is set to "ondemand". Dynamic process management uses the spare server settings to determine when/how to kill processes.

pm.max_requests = 500 The number of request to handle before a child process is killed and respawned. By default, this is set to `0`, meaning unlimited.

You may want a child process to have a limited lifetime. This is useful if you're worried about memory leaks created by your application.

That was a lot about process management! It's important to know, however. In most cases, the default settings are likely too low relative to what your server can handle!

pm.status_path = /status This is off (not set) by default, but you can configure an end-point to get the status of FPM. This example will set a URI end point to `/status`. Once configured, you can use the following URL parameters (assuming you keep it at `/status`) to determine what is output:

- **/status** - Show basic status information
- **/status?full** - Show basic status information + information on each child process
- **/status?full&html** - In HTML format
- **/status?full&xml**- In XML format
- **/status?full&json** - in JSON format

This requires some extra setup. You can't directly query FPM's status via a web request as it "speaks" FastCGI rather than HTTP. We still need a web server to handle the request. In Nginx, this is fairly simple - we can create a "location" block for our server, and limit access for security:

```
# Inside of a Nginx virtual host "server":
location ~ ^/(status|ping)$ {
    access_log off;
    allow 127.0.0.1;
    allow 172.17.42.1;  # A local-only network IP address
    deny all;
    include fastcgi.conf;
    fastcgi_pass 127.0.0.1:9000;  # Assumes using a TCP socket
}
```

The same can be done for Apache:

```
ProxyPass /status fcgi://127.0.0.1:9000/status
ProxyPass /ping fcgi://127.0.0.1:9000/ping
```

If the above configuration looks foreign, it will make more sense after reading the Apache and Nginx chapters.

The above assumes we've set FPM to listen on a TCP socket. You can use the following with Nginx if you are listening on a Unix socket:

```
# Nginx
fastcgi_pass unix:/var/run/php5-fpm.sock;
```

Apache's `ProxyPass` directive is best used with a TCP socket rather than a Unix socket for this use case.

You may have noticed the above configurations also handles a "ping" check. Let's look at that as well:

ping.path = /ping By default, this is not enabled either. This can be used for health checks in a high availability setup. You can have a process periodically ping the FPM pool and ensure it's still functioning. If it's not functioning, the health checker can remove the pool from being used.

ping.response = pong This is simply the text response ("pong") the pool will respond with when pinged. It will respond with a 200 HTTP response code along with the `text/plain` mime type.

access.log The access log for the pool. The default is to not set a log. Usually access logs are set within Nginx or Apache and so it might be redundant for FPM to have them as well.

There are other log options you can set as well, such as the slow log and its time threshold. I suggest checking out the default pool file to find these, I won't cover them in depth here.

chroot Disabled by default, the `chroot` directive is a file path which becomes the relative file path for all of PHP. This includes `php.ini` settings. Chroot is sort of like running PHP-FPM in a jail - the child process can't access other parts of the system. This is a boon for security!

However the downside is ensuring your configuration and application will work with this. *Everything* works relative to this defined file path. This includes paths used for `include`/`require` functions and file upload paths.

By default this is not set, but you should consider using it for security.

If you do use `chroot` in production, mirror this setting in development. Making your development machine match production can be important. You don't want to run into snags due to settings like this when deploying a site to production.

security.limit_extensions = .php By default, Nginx will only parse files with the `.php` extension. If you need PHP-FPM to also process html files which may include PHP, you'll need to uncomment this directive. You can include a `.html` and `.htm` extension here.

Note that this is for directly requested files (usually `index.php`). This is not for files included within other PHP calls. For example, an included/required `.phtml` view file will still work without changing this setting.

You'll also need to adjust the Apache or Nginx configuration to pass off requests for `.html` and `.htm` files to PHP-FPM.

Additional INI directives In Debian/Ubuntu, the same `/etc/php5/fpm/php.ini` file will be used for all resource pools.

Since we can setup separate resource pools per web application, it would be nice to also have separate `php.ini` configurations.

Luckily, we can do that by adding them within our resource pool configuration! The syntax is a little different than a regular `php.ini` file. We can use `php_value` for directives which have a value, or `php_flag` for boolean flags.

For example, the `php.ini` file equivalent to `post_max_size = 16m` would be:

```
1  php_value[post_max_size] = 16m
```

The `php.ini` equivalent to `display_errors = On` would be:

```
1  php_flag[display_errors] = On
```

We can also use `php_admin_value` and `php_admin_flag` directives within the FPM pool configuration. The "admin" version of the flags make it so we can't over-ride them in code using PHP's `ini_set` method. This can be useful for security purposes. For example, if you want to ensure your code cannot turn on error reporting or allow the inclusion of remote files.

Everything Else I've covered most of the available PHP-FPM resource pool configuration. There's still a few you can dig further into.

Server Setup for Multi-Tenancy Apps

For those of you making an application which supports multi-tenancy, here's some web server configurations you might find handy.

A multi-tenancy application is an application which has one code base but supports many organizations/tenants.

This is typical of many SaaS applications where there are tenants under which multiple users can login. A tenant might be a group, company or organization. Each user can be a unique user and log in under one or more tenants.

One common way to divide up tenants within an application is to use subdomains. Beanstalkapp.com[49] uses subdomains in such a way. For example, users of organization "FooBar", would login and operate under the subdomain http://foobar.beanstalkapp.com[50].

To accomplish such a setup, we likely want to use the same code base for the entire application. From a web server point of view, this means we need two things:

1. All subdomains of our application should resolve to our servers
2. A virtual host configured to match any subdomain and pass requests off to our application

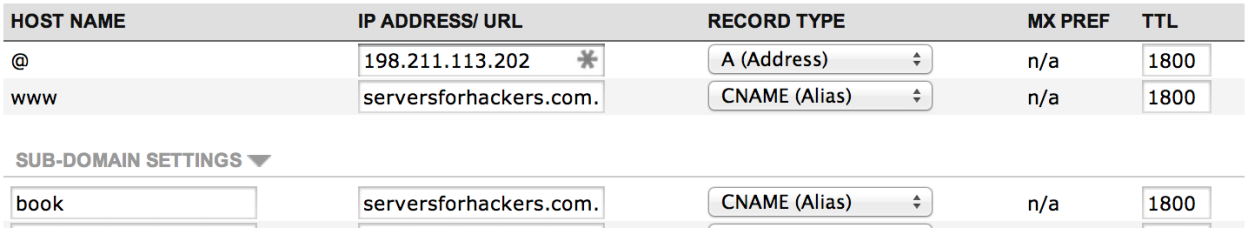
DNS

Before we get to web server configuration, we need to know how this works at a DNS level.

Domains and subdomains can all point to separate servers. Usually we define a root domain, which points to an IP address of a server. Then we can set a subdomain, such as "www" and point it to either the root domain (which in turn points to the server's IP address) or even to another address altogether.

Each subdomain is usually individually specified within the DNS record of a domain. However, we don't want to create an application where we need to go into our domain records and manually enter in a subdomain for every new customer! Instead, we want a wildcard subdomain which matches *any* subdomain.

[49] http://beanstalkapp.com
[50] http://foobar.beanstalkapp.com

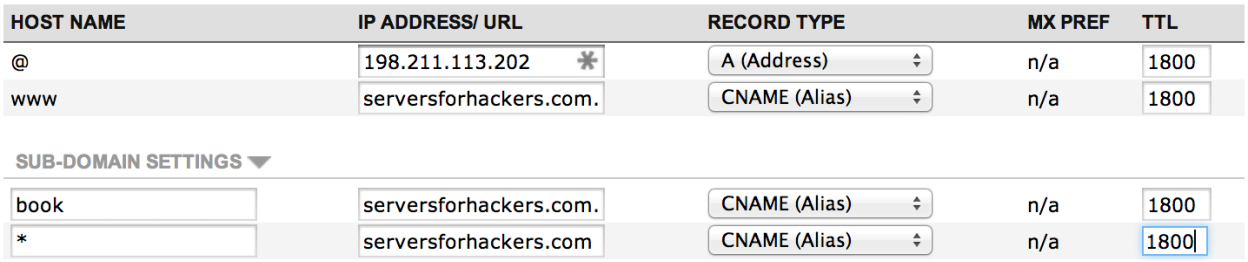


Typical DNS setup. WWW subdomain points to the root domain, which in turn points to a servers IP address.

In the above screenshot, we see the A record for the root domain serversforhackers.com. Then we have a "www" subdomain, defined as a CNAME. This points to the root domain, meaning "resolve to the same IP address as the A record for the root domain."

Then we see another subdomain "book", which also points to the root domain as it is hosted on the same server.

Now we need a DNS record to match *any* subdomain to our server. On DNS providers which support it, we can add a * to denote a wildcard. I use NameCheap:



Any undefined subdomain will get pointed to the application server.

An "A" (Address) record is generally used to point to an IP address. This is always used to define where a root domain points to. We would use an A record for a subdomain if the subdomain needed to point to a different server.

A "CNAME" (Canonical Name) record is generally used to point to a hostname, which in turn will resolve to the IP address of the hostname. This is generally used to point to either the root domain (and thus the same server) or if we want to point to a different server but had a domain/hostname rather than an IP address to use.

Some providers which allow you to set wildcard subdomains include:

- NameCheap
- DNSSimple
- AWS Route53
- Many Others, but not GoDaddy :D

Multi-Tenancy in Apache

Apache lets you create a wildcard subdomain for its `ServerAlias` directive, but not for its `ServerName` directive. As a result, I will typically either create one virtual host for the root domain and "www" subdomain marketing site (assuming it's not part of your main application code) and one virtual host for the application.

```
# Marketing Site
<VirtualHost *:80>

ServerName myapp.io;
ServerAlias www.myapp.io;

DocumentRoot /var/www/marketing-site

</VirtualHost>
```

The above is just a virtual host like any other. It handles `myapp.io` and `www.myapp.io` requests via the `ServerName` and `ServerAlias` directives respectively.

The above virtual host isn't complete, you may need to add in some extra directives as defined in the Apache chapter.

Then we can create a virtual host for the application site:

```
# App Site
<VirtualHost *:80>

ServerName app.myapp.io;
ServerAlias *.myapp.io;

DocumentRoot /var/www/app-site

</VirtualHost>
```

This virtual host handles a base `ServerName` of `app.myapp.io` and then uses `ServerAlias` to match a wildcard subdomain. This directs to a separate `DocumentRoot` than the marketing site.

If your home page (which I'm just assuming might be a marketing page) is part of your application code, than you can use one virtual host:

```
<VirtualHost *:80>

ServerName myapp.io;
ServerAlias *.myapp.io;

DocumentRoot /var/www/myapp

</VirtualHost>
```

You may wish to also use a rewrite rule or a redirect so that people who enter the site via the "www" subdomain get redirected to the root domain. This will prevent confusion if your application attempts to find an organization called "www".

Multi-Tenancy in Nginx

Nginx can have a similar setup with a slightly simpler setup than Apache.

An Nginx virtual host file for the marketing site.

```
# Marketing Site
server {
    listen 80;

    server_name www.myapp.io myapp.io

    root /var/www/marketing-site

}
```

The above server block can be used for a marketing home page. Again, it's just a regular old virtual host for Nginx. Nothing special here - it's setup for `www.myapp.io` and `myapp.io` domains.

This isn't a complete virtual host setup. You likely want to use more directives, which you'll find in the Nginx chapter.

An Nginx virtual host file for the application site.

```
# App Site
server {
        listen 80;

        # Match *.myapp.io
        server_name  ~^(?<user>.+)\.myapp\.io$;

        root /var/www/app-site

        # Optionally pass the subdomain to the app via
        # fastcgi_param, so it's available as an
        # environment variable
        location / {
                include fastcgi.conf; # fastcgi_params for nginx < 1.6.1
                fastcgi_param  USER $user;  # Passing the user to our app!
                fastcgi_pass 127.0.0.1:9000;
        }
}
```

In this server block, we match a wildcard subdomain. As a bonus, we use the RegEx to capture the variable `$user`, which can be passed off to our application using a `fastcgi_pass` directive. This will then become available as an environment variable in our application!

SSL Certificates

SSL Overview

As you're likely aware, being able to send data **securely** over a network (especially a public network) is of growing importance. To that end, many web applications employ the use of SSL certificates to encrypt traffic between a client (often your web browser) and a server (often a web server).

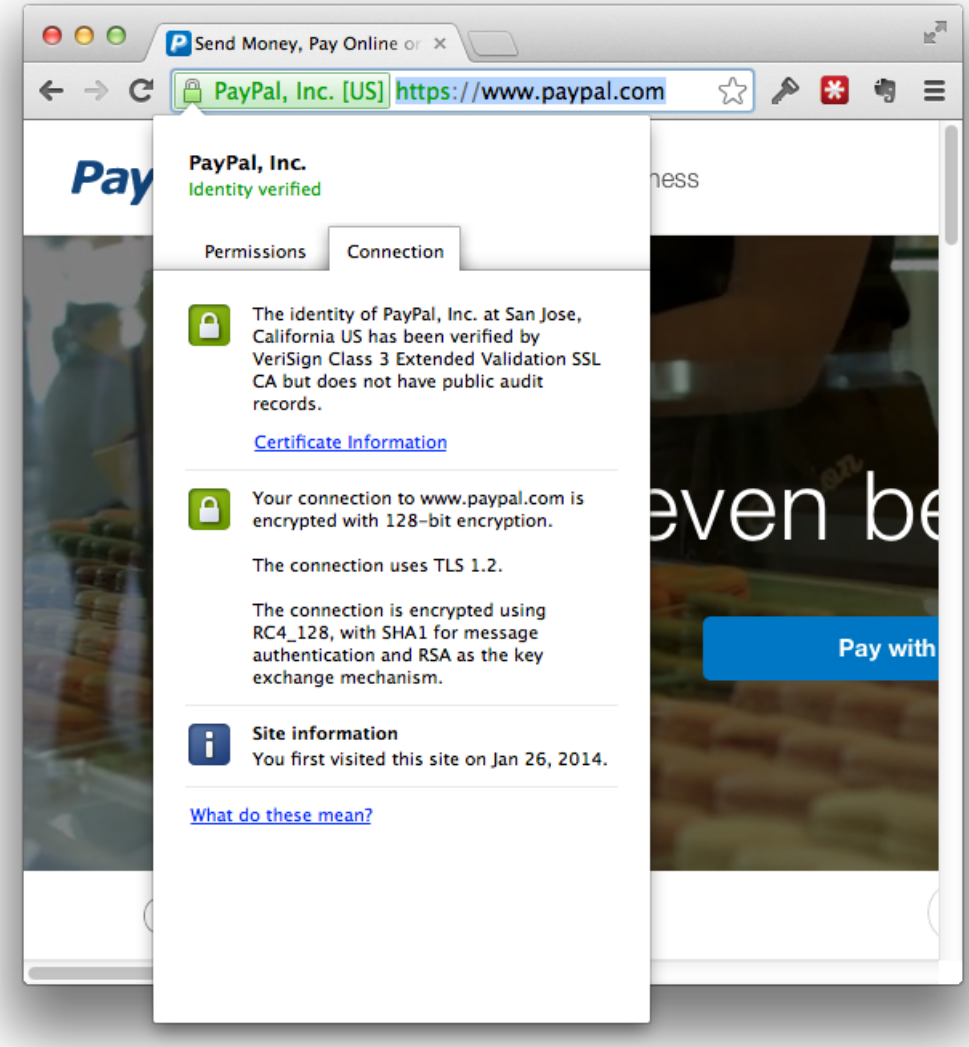
If you're interested in learning more about SSL certificates[51] and the various mechanisms (such as "key certificates", "root certificates", "intermediate certificates" and more), jump to about ~51:45 of this edition of Tech Snap Misconceptions of Linux Security[52].

Using SSL in Your Application

In production, you will have to purchase an SSL certificate. When you purchase an SSL certificate, you are paying for recognized and trusted-third parties (root or intermediate authorities) to say that your SSL certificate is both valid and legitimately used by you, the owner of the certificate. See how PayPal's SSL certificate was verified by VeriSign.

[51] http://en.wikipedia.org/wiki/Secure_Sockets_Layer

[52] http://www.jupiterbroadcasting.com/54142/misconceptions-of-linux-security-techsnap-155/



PayPal SSL Certificate

SSL certificates affect your application. When a site is visited using "https", browsers expect all resources (images, javascript, css and other static assets) to also be linked to and downloaded using "https" as well. Otherwise browsers either don't load the assets and show scary warning messages to your users. This means you need to be able to serve your assets *and any third party assets* with "https". Third party assets are any not directly on your web server (images hosted on your CDN of choice, for example).

That means it is useful to have a way to test your applications with an SSL certificate in development, instead of waiting for your site to launch to find issues.

Creating Self-Signed Certificates

Unless there are some extenuating circumstances, you shouldn't need to buy an SSL certificate for use in development. Instead, you can create a "self-signed" certificate, which will work in your local computer. Your browsers will initially give you a warning for using an un-trusted certificate, but you can click passed that and test your web application with your own SSL certificate.

The basic steps to create a self-signed certificate are:

1. Create a Private Key
2. Create a Certificate Signing Request (CSR)
3. Create a Self-Sign certificate using the Private Key and the CSR
    - Alternatively, if you purchased an SSL, the last step is accomplished by the certificate signing authority
4. Install the certificate for use on your web server

To start, first make sure you have OpenSSL installed. Most flavors of Linux have this "out of the box", but you should be able to easily install it if not:

```
# Check if openssl is installed
$ which openssl
/usr/bin/openssl

# Or, if no output from command `which`:
sudo apt-get install openssl
```

Heartbleed

There was a nasty OpenSSL vulnerability you may have heard about: The Heartbleed Bug[53]. OpenSSL 1.0.1 through 1.0.1f (inclusive) are vulnerable. This means 1.0.1g and greater are fixed. You should see if any of your servers need updating. You can use http://filippo.io/Heartbleed/[54] to test if your site is vulnerable.

Note that you may have OpenSSL version 1.0.1f installed which contains the Heartbleed fix. Ubuntu, like other distributions, often backports security fixes rather than update the actual software. Run `apt-cache show openssl | grep Version` to ensure you have `1.0.1f-1ubuntu2` or `1.0.1f-1ubuntu2.5`, both of which contain the fix to Heartbleed. Ubuntu 14.04 should not be vulnerable.

[53] http://heartbleed.com/

[54] http://filippo.io/Heartbleed/

We need a place to put our certificates. I usually put them in the `/etc/ssl` directory, which contains other system certificates. For any new certificate, I create a new directory. If we're creating an SSL for `example.com`, create the directory `/etc/ssl/example`.

Once we have a directory created, we can begin creating our certificate. First, we need a private key:

```
# Create a 2048 bit private key
# Change your -out filepath as needed
sudo mkdir -p /etc/ssl/example
sudo openssl genrsa -out "/etc/ssl/example/example.key" 2048
```

The private key is used to generate our Certificate Signing Request (CSR) and is needed to properly sign/create our certificate. It's also used to properly decrypt SSL traffic.

Next we need to create the CSR. The CSR holds information used to generate the SSL certificate. The information provided also contains information about the company or entity owning the SSL.

Generating a CSR uses the Private Key we previously created:

```
sudo openssl req -new -key "/etc/ssl/example/example.key" \
                         -out "/etc/ssl/example/example.csr"
```

This will ask you the following series of question:

CSR generating questions and my responses

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Connecticut
Locality Name (eg, city) []:New Haven
Organization Name (eg, company) [Internet Widgets Pty Ltd]:Fideloper
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:example.local
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

The `Common Name` option is the most important, as your domain used with the certificate needs to match it. If you use the "www" subdomain for your site, this means **specifying the "www" subdomain in the Common Name field as well!**

I left some fields blank. You can skip **Organizational Unit Name** and **Email Address** for a self-signed certificate. I also choose to leave the "optional company name" field blank. Finally, I elected NOT to add in a challenge password[55]. This is something used by the certificate authority (if you're purchasing a certificate) in the case you need to regenerate a certificate. Your web server may also require this password when restarting as well.

So, we now have `example.key` and `example.csr` files created. Let's finish this up by creating the self-signed certificate.

```
1 sudo openssl x509 -req -days 365 -in "/etc/ssl/example/example.csr" \
2                                   -signkey "/etc/ssl/example/example.key" \
3                                   -out "/etc/ssl/example/example.crt"
```

Here's what we did:

- `sudo openssl x509` - Create an SSL certificate following x509 specification[56]
- `-req` - State that we're generating a certificate
- `-days 365` - This certificate is valid for one year
- `-in "/etc/ssl/example.csr"` - The CSR generated for this certificate
- `-signkey "/etc/ssl/example.key"` - The Private Key used for this certificate
- `-out "/etc/ssl/example.crt"` - Where to put the new certificate file

Great, our self-signed certificate for `example.com` is created! We'll cover installing it into our web servers in just a bit.

Creating a Wildcard Self-Signed Certificate

I use the Xip.io service so that I can avoid editing my hosts file for local development servers. I've found it useful to automate the process of creating a self-signed *wildcard* xip.io certificate for my local servers to test my local sites under SSL.

Here we'll see how to create a wildcard subdomain SSL certificate. I'll also show you how to do it in a way that can be automated, eliminating the need for human interaction.

Let's begin! Create a new shell script and call it `generate-ssl.sh`:

[55] http://serverfault.com/questions/266232/what-is-a-challenge-password

[56] http://en.wikipedia.org/wiki/X.509

Automating the creation of a self-signed certificate

```
#!/usr/bin/env bash

# Specify where we will install
# the xip.io certificate
SSL_DIR="/etc/ssl/xip.io"

# Set the wildcarded domain
# we want to use
DOMAIN="*.xip.io"

# A blank passphrase
PASSPHRASE=""

# Set our CSR variables
SUBJ="
C=US
ST=Connecticut
O=
localityName=New Haven
commonName=$DOMAIN
organizationalUnitName=
emailAddress=
"

# Create our SSL directory
# in case it doesn't exist
sudo mkdir -p "$SSL_DIR"

# Generate our Private Key, CSR and Certificate
sudo openssl genrsa -out "$SSL_DIR/xip.io.key" 2048

sudo openssl req -new -subj "$(echo -n "$SUBJ" | tr "\n" "/")" \
                 -key "$SSL_DIR/xip.io.key" \
                 -out "$SSL_DIR/xip.io.csr" -passin pass:$PASSPHRASE

sudo openssl x509 -req -days 365 -in "$SSL_DIR/xip.io.csr" \
                  -signkey "$SSL_DIR/xip.io.key" \
                  -out "$SSL_DIR/xip.io.crt"
```

The above script follows all of our previous steps, except it does some fancy bash scripting so we can automate passing in the CSR generating variables using the `-subj` flag and some string parsing.

Once that's saved, you can run script with the following command:

```
sudo bash generate-ssl.sh
```

Then you can see those generated files in `/etc/ssl/xip.io/`.

Note that we defined the domain as `*.xip.io.`. We signified the use of a wildcard subdomain with the * character. This will let us use any subdomain. Otherwise this mirrors the process we did above when "manually" creating our SSL certificate for the `example.com` domain.

Now that we've generated some certificates, let's see how to use them in our favorite web servers.

Apache Setup

The first thing to do in Apache is to make sure `mod_ssl` is enabled. On Debian/Ubuntu, you can do this via:

```
# Enable SSL module
sudo a2enmod ssl
# Then restart:
sudo service apache2 restart
```

Next, we need to modify our vhost to accept https traffic on port `443`.

Up until now, we've create a vhost to listen on port `80`, the default http port. That might look like this:

File: /etc/apache2/sites-available/example.conf

```
<VirtualHost *:80>
        ServerName example.local

        DocumentRoot /var/www/example.local

        ... and the rest ...
</VirtualHost>
```

To enable SSL for this site, we can create another vhost file, or add another block to our `example.conf` file. For example, the following new vhost file will listen on port `443`, the default https port:

File: /etc/apache2/sites-available/example-ssl.conf

```
<VirtualHost *:443>
        ServerName example.local

        DocumentRoot /var/www/example.local

        SSLEngine on

        SSLCertificateFile  /etc/ssl/example/example.crt
        SSLCertificateKeyFile /etc/ssl/example/example.key

        # And some extras, copied from Apache's default SSL conf virtualhost
        <FilesMatch "\.(cgi|shtml|phtml|php)$">
            SSLOptions +StdEnvVars
        </FilesMatch>

        BrowserMatch "MSIE [2-6]" \
            nokeepalive ssl-unclean-shutdown \
            downgrade-1.0 force-response-1.0
        # MSIE 7 and newer should be able to use keepalive
        BrowserMatch "MSIE [7-9]" ssl-unclean-shutdown

        ... and the rest ...
</VirtualHost>
```

Note that the above vhost's were not complete - you'll need to fill in some extra parameters and directives from the Apache chapter. The above examples are simply for setting up the SSL certificates.

By default, Debian/Ubuntu installs of Apache use `BrowserMatch "MSIE [17-9]"` in their example SSL file. This is a "bug"[57], the value should be: `BrowserMatch "MSIE [7-9]"`.

And that's it! Once you have that in place and enabled, you can reload Apache's config (`sudo service apache2 reload`) and try it out!

If you are using a self-signed certificate, you'll still need to click through the browser warning saying the Certificate is not trusted.

[57] https://bugs.launchpad.net/ubuntu/+source/apache2/+bug/626728

Apache & Xip.io

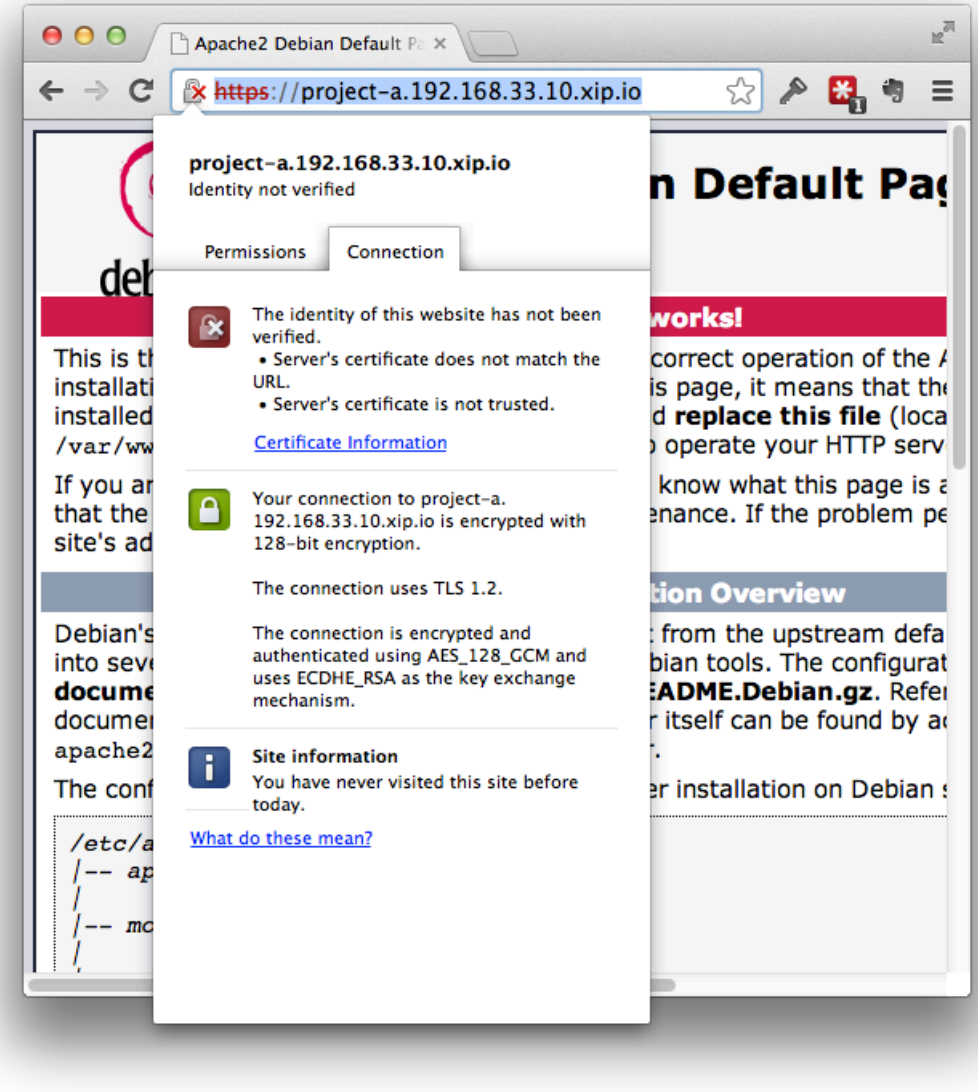
Let's see what that looks like for the wildcard xip.io setup. The following virtualhost is for a web app located at `project-a.192.168.33.10.xip.io`, where "192.168.33.10" is the IP address of the server.

File: /etc/apache2/sites-available/example-ssl.conf

```
<VirtualHost *:443>
        ServerName project-a.192.168.33.10.xip.io

        DocumentRoot /var/www/example.local

        SSLEngine on

        SSLCertificateFile  /etc/ssl/xip.io/xip.io.crt
        SSLCertificateKeyFile /etc/ssl/xip.io/xip.io.key

        # And some extras, copied from Apache's default SSL conf virtualhost
        <FilesMatch "\.(cgi|shtml|phtml|php)$">
            SSLOptions +StdEnvVars
        </FilesMatch>

        BrowserMatch "MSIE [2-6]" \
            nokeepalive ssl-unclean-shutdown \
            downgrade-1.0 force-response-1.0
        # MSIE 7 and newer should be able to use keepalive
        BrowserMatch "MSIE [7-9]" ssl-unclean-shutdown

        ... and the rest ...
</VirtualHost>
```

After that's setup, you can reload your Apache config (Debian/Ubuntu: `sudo service apache2 reload`) and test it out!

Here we can see the SSL certificate working, but of course our browser doesn't trust it since it's not certified by a trusted third party. That's fine though, we can still test our application's use of SSL by clicking through the warnings.



Xip.io wildcard subdomain via self-signed certificate

"Server's certificate does not match the URL."

You might see this error message when viewing your site under a self-signed certificate with our Xip.io address. It turns out that matching wildcards isn't supported the same across implementations/browsers, especially the "sub-sub-sub-domains" we use with xip.io.

It will still function fine for development purposes, however. Read here[58] for some more information.

[58] http://www.hanselman.com/blog/SomeTroubleWithWildcardSSLCertificatesFireFoxAndRFC2818.aspx

Nginx Setup

For Nginx, we typically have a server "block" listening on port 80 (the default port for http). This will look something like this:

File: /etc/nginx/sites-available/example

```
server {
        listen 80 default_server;

        server_name example.local;
        root /var/www/example.com;

        ... and the rest ...
}
```

For setting up an SSL, we want to listen on port 443 (a default port for https) instead:

File: /etc/nginx/sites-available/example-ssl

```
server {
        listen 443;
        root /var/www/example.com;

        ... and the rest ...
}
```

These server blocks can be in the same configuration file or in separate ones. That's completely up to you. Just remember to symlink any configuration files to the `/etc/nginx/sites-enabled` directory if they need to be enabled.

To setup the https server block with our SSL certificate, we just need to add a few lines:

File: /etc/nginx/sites-available/example-ssl

```
server {
        listen 443 ssl;

        server_name example.local;
        root /var/www/example.com;

        ssl_certificate     /etc/ssl/example/example.crt;
        ssl_certificate_key /etc/ssl/example/example.key;
```

```
9
10          ... and the rest ...
11  }
```

And that's it! Once you have that in place and enabled, you can reload Nginx (`sudo service nginx reload`) and try it out!

> If you are using a self-signed certificate, you'll still need to click through the browser warning saying the Certificate is not trusted.

Nginx & Xip.io

Similar to the Apache setup, for using `xip.io` you can adjust the `server_name` and certificate paths and be on your way:

File: /etc/nginx/sites-available/xipio

```
1   server {
2           listen 443 ssl;
3
4           server_name project-a.192.168.33.10.xip.io;
5           root /var/www/projecta;
6
7           ssl_certificate     /etc/ssl/xip.io/xip.io.crt;
8           ssl_certificate_key /etc/ssl/xip.io/xip.io.key;
9
10          ... and the rest ...
11  }
```

Once Nginx is reloaded, this will work as well! Don't forget to fill in the rest of the virtual host configuration as per the Nginx web server chapter..

One Server Block

As per the Nginx Admin Guide[59], you can define both http and https in one server block:

[59] http://nginx.com/admin-guide/nginx-ssl-termination

File: /etc/nginx/sites-available/xipio

```
server {
        listen 80;
        listen 443 ssl;

        server_name project-a.192.168.33.10.xip.io;
        root /var/www/projecta;

        ssl_certificate     /etc/ssl/xip.io/xip.io.crt;
        ssl_certificate_key /etc/ssl/xip.io/xip.io.key;

        ... and the rest ...
}
```

Extra SSL Tricks

Here's a one-liner for generating an self-signed certificate in one go:

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
                 -keyout /etc/ssl/example/example.key \
                 -out /etc/ssl/example/example.crt
```

This will ask you for the same options as about to generate the CSR and then generate the certificate automatically.

Some explanation of the command:

- `sudo openssl req` - Req says we're generating a certificate
- `-x509` - Using the x509 specification[60]
- `-nodes` - Since we're doing this in one step, don't encrypt the Private Key (since it may require a password). Read more here[61].
- `-days 365` - Create a certificate valid for 1 year
- `rsa:2048` - Use a 2048 bit Private Key
- `-keyout /etc/apache2/ssl/example.key` - Where to put the Private Key
- `-out /etc/apache2/ssl/example.crt` - Where to put the generated Certificate

To use `curl` with your site when using a self-signed certificate, you need to tell it not to verify the SSL connection. Here's how to use `curl` with a self-signed certificate:

```
curl --insecure https://myapp.local

# -K is equivalent to --insecure:
curl -K https://myapp.local
```

We need to do the same when using `wget` with a self-signed certificate:

```
wget --no-check-certificate https://myapp.local/somefile
```

[60] http://en.wikipedia.org/wiki/X.509

[61] http://stackoverflow.com/questions/5051655/what-is-the-purpose-of-the-nodes-argument-in-openssl

Multi-Server Environments

As servers become more and more a commodity, they are both cheaper and easier to get. As a result, we often see setups where we create and use multiple servers.

This is often born from necessity. Many cloud hosting providers provide many, smaller servers rather than fewer, powerful ones. In these cases, we're essentially forced into creating distributed architectures to keep our sites fast.

Building in reliability also requires multi-server environments. High Availability is accomplished through distributed systems, monitoring and automation.

We've covered some of the knowledge needed to accomplish a high availability setup, such as the need to manage firewalls so the servers can communicate to each other. However we still need to know some more!

In the following chapters, we'll get in depth on multi-server environments, with a focus on load balancing.

Implications of Multi-Server Environments

When you put your web application behind a load balancer, or any type of reverse proxy[62], you immediately need to take some important factors into consideration.

This chapter will cover those considerations, as well as discuss common solutions.

Asset Management

Using a load balancer implies that you have more than one server processing requests. In this situation, how you manage your static assets (images, JS, CSS files) becomes important.

Imagine a scenario where an image lives on one web server, but not the other. In this situation, a user of your application will see a 404 response when the load balancer tries to pull the image from the web server which does not have the image.

This is a common issue when your application accepts user uploads (perhaps with a CMS). User-uploaded files can't simply live on the web server they were uploaded to. When an uploaded `jpg` file only lives on one web server, a request for that image will result in a 404 response when the load balancer attempts to find it on web server which does *not* have the image!

In a distributed environment, one often (somewhat ironically) needs to centralize! In this case, the web servers need to have a common file store they can access.

One way this is done is via a shared network drive (NAS[63], for example). This, however, gets slow when there are many files or high levels of traffic. Furthermore, if your architecture is distributed across several data centers, then a shared network drive can become too slow; Your web servers would be too far away from them and the network latency too high.

Central File Store

A common (and better) solution is to host all static assets (user-uploaded and otherwise) in a separate, central location, such as Amazon's S3.

> Within Amazon, this can be taken a step further. An S3 bucket can be integrated with their CDN CloudFront. Your files can then be served via a true CDN. You may also wish to use other CDN's such as CloudFlare or MaxCDN directly.

[62] http://en.wikipedia.org/wiki/Reverse_proxy

[63] http://en.wikipedia.org/wiki/Network-attached_storage

For your static assets, you can use automated tools such as Grunt or Gulp to automate these tasks for you. For example, you can have Grunt watch your files for changes, minify and concatenate CSS, JS and images, generate production-ready files, and then upload them to a central location.

For user-uploaded content, you'll likely need to do some coding around uploading files to a temporary location, and then sending them off to S3 via AWS's API.

Environment-Based URLs

You often don't use central file stores in development. This means you likely have local static assets rather than remote assets stored in S3, a CDN or similar. This means in development, you need to be able to serve your static files locally.

One thing I do on projects is to change the URL of assets based on the environment. Using a helper function of some sort, I'll have code output the development machine's URL to HTML so the static assets are loaded locally.

In production, this helper will output URLs for your file-store or CDN of choice. Combined with some automation (perhaps with Grunt or Gulp), this gives you a fairly seamless workflow between development and production.

Sessions

Similarly to the issue of Asset Management, how you handle sessions becomes an important consideration. Session information is often saved on a temporary location within a web server's file system. A user may log in, creating a session on one web server. On a subsequent request, however, the load balancer may bounce that user to another web server which doesn't have that session information. The client would think they were forcefully logged out.

There are two common fixes for this session scenario.

Sticky Sessions

The first is to set your load balancer to use "sticky sessions", often also called "session affinity". This will take one client and always route their request to the same web server. This let's the web server keep its default behavior of saving the session locally, leaving it up to the load balancer to get a client back to that server. This can skew the sharing of work load around your web servers a bit.

Shared Storage

The second fix for this is to use a central session store. Typical storage used for sessions are fast in-memory stores such as Redis or Memcached. Persistent stores such as a database are also commonly used, but aren't recommended for high-volume sites.

Since session data does not necessarily need to be persistent, and can have lots of traffic, a central in-memory data store may be preferred. In this architecture, all the web servers connect to a server working as the central session store, growing your infrastructure a bit, but letting your work load be truly distributed.

Lost Client Information

Closely related to the session issue is detecting *who* the client is. If the load balancer is a proxy between a client and your web application, it might appear to your web servers and application that *every* request is coming from the load balancer! Your application wouldn't be able to tell one client from the other.

Luckily, most load balancers provide a mechanism for giving your application this information. If you inspect the headers of a request received from a load balancer, you might see some of these included:

- `X-Real-Ip`
- `X-Forwarded-For`
- `X-Forwarded-Proto`
- `X-Forwarded-Port`
- `X-Forwarded-Scheme`

These headers can tell you the client's IP address, the schema used (http vs https) and which port the client made the request on. If these are present, your application's job is to sniff these headers out and use them in place of the usual client information.

IP Addresses & Ports

Having an accurate IP address of a client may be important to your application. Some applications use the client's IP address to perform functions such as rate limiting or metric gathering. Furthermore, having a client's IP address can help identify malicious traffic patterns when inspecting logs.

The `X-Forwarded-For` (or `X-Real-Ip`) header, which should include the client's IP address, can be used by your application if the header is found in the HTTP request. This is assuming the source of the proxied web request is trusted - some measure should be taken to ensure all requests come from your load balancer instead of from external sources which may have malicious intent.

Protocol/Schema and Port

Knowing the protocol (http, https) and port used by the client is also important. If the client is connecting over an SSL (with a https url), that encrypted connection might *end* at the load balancer.

The load balancer would then send an "http" request to the web servers. This means the web servers will receive the traffic over "http" instead of "https".

Many frameworks attempt to guess the site address based on the request information. If your web application is receiving a "http" request over port 80, then any URLs it generates or redirects it sends will likely be on the same protocol. This means that a user might get redirected to a page with the wrong protocol or port when behind a load balancer!

Sniffing out the `X-Forwarded-Proto`, `X-Forwarded-Scheme` and/or `X-Forwarded-Port` header then becomes important so that the web application can generate correct URLs for redirects or for printing out URLs in templates (think form actions, links to other site pages and links to static assets).

Trusted Proxies

Many frameworks, including Symfony and Laravel, can handle sniffing out the `X-Forwarded-*` headers for you. However, they may ask you to configure a "trusted proxy[64]". If the request comes from a proxy who's IP address is trusted, then the framework will seek out and use the `X-Forwarded-*` headers in place of the usual mechanisms for gathering that information.

This provides a very nice abstraction over this HTTP mechanism, allowing you to forget this is a potential issue while coding!

However, you may not always know the IP address of your load balancers. This is the situation when using some Cloud-provided load balancers, such as Rackspace's load balancer or AWS's Elastic Load Balancer. In these situations, you must set your application to trust all proxies and their `X-Forwarded-*` headers.

SSL Traffic

In a load balanced environment, SSL traffic is typically decrypted at the load balancer. Web traffic is then sent to the web servers as "http" rather than "https". This is the most common approach.

However, there's actually a few ways to handle SSL traffic in a distributed environment.

SSL Termination

When the load balancer is responsible for decrypting SSL traffic before passing the request on, it's referred to as "SSL Termination". In this scenario, the load balancer alleviates the web servers of the extra CPU processing needed to decrypt SSL traffic. It also gives the load balancer the opportunity to append the `X-Forwarded-*` headers to the request before passing it onward.

The downside of SSL Termination is that the traffic between the load balancers and the web servers is *not* encrypted. This leaves the application open to possible man-in-the-middle attacks. However, this

[64] https://github.com/fideloper/TrustedProxy

is a risk usually mitigated by the fact that the load balancers are often within the same infrastructure (data center) as the web servers. Someone would have to get access to traffic between the load balancers and web servers by being within the data-centers internal network (possible, but less likely), or by gaining access to a server within the infrastructure.

SSL Pass-Through

Alternatively, there is "SSL Pass-Through". In this scenario, the load balancer does not decrypt the request, but instead passes the request through to a web server. The web server then must decrypt it.

This solution costs the web servers more CPU cycles, but this load is distributed amongst the web servers rather than centralized at the load balancer(s).

You also may lose some extra functionality that load-balancing proxies can provide, such as DDoS protection. However, this option is often used when security is an important concern, as the traffic is encrypted until it reaches its final destination.

SSL Pass-Through prevents the addition of the `X-Forwarded-*` headers. Load balancers implementing SSL Passthru need to operate at the TCP level rather than HTTP, as they can't unencrypt the traffic to inspect and identity the traffic as a HTTP request.

This means applications which need to know the client's IP address or port will **not** receive it. Therefore, the needs of your application may determine where an SSL connection is terminated.

Both

A third, less common, option is to use both styles of SSL encryption. In this scenario, the load balancer unencrypts the SSL traffic and then adjusts the HTTP request, adding in the `X-Forwarded-` headers or applying any other rules. It then sends the request off to the web servers as a **new** HTTP request!

Amazon AWS load balancers give you the option of generating a (self-signed) SSL for use between the load balancer and the web servers, giving you a secure connection all around. This, of course, means more CPU power being used, but if you need the extra security due to the nature of your application, this is an great option.

Note that when communicating between your load balancer and web servers, it's perfectly OK to use self-signed certificates. Only the public-facing SSL certificates need be purchased. You may want to set up your own private certificate authority when doing so, however, so that your applications don't bulk at sending traffic to untrusted SSL certificates. This will allow you to make your code trust your self-signed certificates.

Logs

So, now you have multiple web servers, but each one generates their own log files! Going through each servers' logs is tedious and slow. Centralizing your logs can be very beneficial.

The simplest ways I've done this is to combine Logrotate's functionality with an uploaded to an S3 bucket. This at least puts all the log files in one place that you can look into. This covered in the Logging chapter.

However, there's **plenty** of centralized logging servers that you can install in your infrastructure or purchase. The SaaS offerings in this arena are often easily integrated, and usually provide extra services such as alerting, search and analysis.

Some popular self-install loggers:

- LogStash[65]
- Graylog2[66]
- Splunk[67]
- Syslog-ng[68]
- Rsyslog[69]

Some popular SaaS loggers:

- Loggly[70]
- Splunk Storm[71]
- Paper Trail[72]
- logentries[73]
- BugSnag[74] - Captures errors, not necessarily all logs

[65] http://logstash.net/
[66] http://graylog2.org
[67] http://www.splunk.com
[68] https://wiki.archlinux.org/index.php/Syslog-ng
[69] https://wiki.archlinux.org/index.php/rsyslog
[70] https://www.loggly.com
[71] https://www.splunkstorm.com
[72] https://papertrailapp.com
[73] https://logentries.com
[74] https://bugsnag.com

Load Balancing with Nginx

Nginx can do more things than act as a web server. One of its other popular uses is to act as an HTTP load balancer.

Here we'll cover how to use Nginx as a Load Balancer before moving onto the more fully-featured HAProxy.

Balancing Algorithms

One consideration when load balancing is configuring how you'd like the traffic to be distributed. Load balancers often provide a variety of algorithms for load balancing. Nginx offers the following strategies:

- **Round Robin** - Nginx chooses which server will fulfill a request in order they are defined. This is the default, which is used if no strategy is explicitly defined. Round Robin is a good "default" if you're unsure which suits your needs.
- **Least Connections** - Request is assigned to the server with the least connections (and presumably the lowest load). This is best for applications with relatively long connections, perhaps those using web sockets, server push, long-polling or HTTP request with long keep-alive parameters.
- **Ip-Hash** - The Client's IP address is hashed. The resulting hash is used to determine which server to send the request to. This also effectively makes user sessions "sticky". Subsequent requests from a specific user always get routed to the same server. This is one way to get around the issue of user sessions behaving as expected in a distributed environment. Hashes are common if the load balancer is used as a cache server - if there are multiple cache servers, this can result in a higher cache hit rate.
- Generic **Hash** - A user-defined key can be used to distribute requests across upstream servers.

IP-Hash is not the only way to accomplish session stickiness ("session affinity"). You can also use a `sticky` directive, which will tell Nginx what cookie to read to determine which server to use. That will be covered below.

Weights

With all but Round Robin algorithm, you can assign weights to a server. Heavier-weighted servers are more likely to be selected to serve a request. This is good if your stack has servers with uneven amounts of resources - you can assign more requests to powerful servers. Another use case might be to test application or server functionality - you can send small amounts of traffic to the server with the experimental software and gauge its effectiveness before pushing it fully into production.

Configuration

Let's say we have three NodeJS processes running, each listening for HTTP requests. If we want to distribute requests amongst them. We can configure our Nginx to proxy HTTP requests to the defined upstream servers (the NodeJS processes) like so:

File: /etc/nginx/sites-available/example - Example load balancing configuration

```
# Define your "upstream" servers - the
# servers request will be sent to
upstream app_example {
    least_conn;
    server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9002 max_fails=3 fail_timeout=30s;
}

# Define the Nginx server
# This will proxy any non-static directory
server {
    listen 80;
    listen 443 ssl;
    server_name example.com www.example.com;

    access_log /var/log/nginx/example.com-access.log;
    error_log  /var/log/nginx/example.com-error.log error;

    # Browser and robot always look for these
    # Turn off logging for them
    location = /favicon.ico { log_not_found off; access_log off; }
    location = /robots.txt  { log_not_found off; access_log off; }

    # You'll need to have your own certificate and key files
    # This is not something to blindly copy and paste
    ssl_certificate     /etc/ssl/example.com/example.com.crt;
    ssl_certificate_key /etc/ssl/example.com/example.com.key;

    # Handle static files so they are not proxied to NodeJS
    # You may want to also hand these requests to another upstream
    # set of servers, as you can define more than one!
    location / {
        try_files $uri $uri/ @proxy;
    }
```

```
36
37      # pass the request to the node.js server
38      # with some correct headers for proxy-awareness
39      location @proxy {
40              include proxy_params;
41              proxy_set_header X-Forwarded-Port $server_port;
42
43          proxy_pass http://app_example/;
44          proxy_redirect off;
45
46          # Handle Web Socket connections
47          proxy_http_version 1.1;
48          proxy_set_header Upgrade $http_upgrade;
49          proxy_set_header Connection "upgrade";
50      }
51  }
```

There's quite a bit going on here! We'll go over each section next.

Upstream

First, we defined the "upstream" block. This setup will proxy requests to the three NodeJS processes which are setup to accept HTTP requests and respond to them.

```
1  upstream app_example {
2      least_conn;
3      server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
4      server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
5      server 127.0.0.1:9002 max_fails=3 fail_timeout=30s;
6  }
```

Here we use the `least_conn` balancing strategy. Generally I choose this or round-robin. Defining no balancing algorithm will default to round-robin.

Then our three NodeJS servers are defined. Thse happen to be listening on localhost (127.0.01), but in production, these will not necessarily be locally running listeners.

A typical setup would be to have other applications/servers listening for connections on their own servers, usually over a private (not exposed to the public internet) network. Note that you can use hostnames as well as unix sockets (e.g. `unix:/path/to/socket`) as well, instead of IP addresses.

Passive Health Checks

The above configuration uses some basic ("passive") health checks. We set the `max_fails` directive, which is the maximum number of times a server can be unresponsive before Nginx stops sending traffic to that server. We also define `fail_timeout`, which is the amount of time a server will be considered "failed" before trying to send traffic to it again. The `fail_timeout` directive will **also** determine within how many seconds the `max_files` can be reached before the count is reset. This is double duty, always set to the same number of seconds.

Active Health Checks

Nginx also has an "active" health check. Within the not-yet-discussed `location` block, we can add the `health_check` directive to the `location` block. This will check the base url "/" for each of our servers every 5 seconds. If a communication error occurs, a timeout is reached, or an HTTP response of 400 and greater occurs, the health check will fail the proxied server, taking it out of the rotation.

Setting the `passes` parameter will tell Nginx that it needs to pass that many consecutive times before being considered healthy again. You can also optionally set a URL to check via the `uri` parameter.

The `health_check` parameter requires the use of the `zone backend 64k;` directive in the `upstream` block. This configuration sets 64k bits of shared memory for Nginx's processes to use to track the status of defined upstream servers.

An abbreviated look at how an active health check would look:

```
upstream app_example {
    zone backend 64k;
    least_conn;
    server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9002 max_fails=3 fail_timeout=30s;
}

server {
        # Other items omitted...

        location @proxy {
            health_check interval=5 fails=3 passes=2 uri=/some/path;
            include proxy_params;

            # Other items omitted...
        }
}
```

Session Affinity

I always suggest writing your applications so they don't require session affinity. This can usually be accomplished by using cookie-based sessions storage or by using a central store that can be used to store sessions, such as redis or memcached. This is discussed in a previous chapter which outlines what to be aware of under a distributed environment.

If you need or prefer to have your load balancer send users to the save server in all cases (using session affinity), you can! To do so, we can use the `sticky` directive:

```
upstream app_example {
    zone backend 64k;
    least_conn;
    sticky cookie srv_id expires=1h;
    server 127.0.0.1:9000 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9001 max_fails=3 fail_timeout=30s;
    server 127.0.0.1:9002 max_fails=3 fail_timeout=30s;
}
```

With this directive, Nginx will first select a server to send the traffic to when it receives a connection without a set cookie. It will then insert a cookie into the *response* from that server. A client (such as a browser) will save that cookie and return it in subsequent requests, which Nginx will use to map the request to the same server as the original request.

Our applications can safely ignore the existence of this cookie, which I've named `srv_id` above.

Server

Next let's look at the `server` block. A lot of this is simple boiler plate explained in the Nginx chapter. The interesting things are the two `location` blocks.

The first block, as explained in the Nginx chapter, will attempt to find a matching static file or directory. Failing that, it will send requests to the `location` block labeled `@proxy`, for the load balancer to proxy.

This is a decision used just for demonstration. This example happens to proxy application requests only. We can, and often will want to, have requests for static assets also proxied to load balanced (upstream) servers.

Then we get to our more interesting `location` block - the one used to proxy requests for load balancing:

```
location @proxy {
      health_check;
      include proxy_params;
      proxy_set_header X-Forwarded-Port $server_port;

      proxy_pass http://app_example/;

      # Handle Web Socket connections
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
}
```

Proxy Params

We've discussed the `health_check` directive already. Let's go onto `include proxy_params`. This includes the file `/etc/nginx/proxy_params`, which has some proxy boiler plate we might want:

- Sets the `Host` header to the original request
- Sets the `X-Real-Ip` header to the client's IP address
- Sets the `X-Forwarded-For` header to the client IP's address (same as above, but the two headers are often used differently)
- Sets the `X-Forwarded-Proto` header to the scheme used by the client (http or https)

In addition to the above, I like to add the `X-Forwarded-Port` header, so our web applications can redirect to the proper port, if a non-standard one is used. I set this to `$server_port`, so it adjusts based on if the request is received from an HTTP (port 443) connection or not.

Since 80 and 443 are standard ports, the `X-Forwarded-Scheme` header is usually enough for any backend application to use to send redirect responses, but if a non-standard port was ever listened on by Nginx, we can have the application rely on `X-Forwarded-Port` instead.

Proxy Pass

We have two `proxy_*` settings here. First, `proxy_pass` simply says to pass the request to our defined backend `app_example`.

One option to explore (and left to its defaults here) is `proxy_redirect`. This can do some interesting things.

Note above that we inject some headers into each request to our proxied servers (client IP address and port). This is done so our application can do things like redirect to correct ones. The `proxy_redirect` directive can help here as well, especially in cases where our application doesn't properly redirect for us.

SSL Support

This setup uses SSL Termination. In this setup, we've setup the server block to listen on SSL's port 443 in addition to port 80. The SSL request is decrypted at the Nginx server before the request is sent (unencrypted at port 80) to the proxied servers. This is why it's called "SSL Termination" - the SSL connection is terminated at the Load Balancer.

The opposite is SSL Pass-Thru, in which the SSL connection is passed onto the proxied servers without being unencrypted. Unlike HAProxy, which we'll discuss in an upcoming chapter, Nginx cannot do SSL Pass-Thru. SSL Pass-Thru must be done at the lower TCP layer, however Nginx only operates on the higher-level HTTP.

How to setup an SSL connection within Nginx and Apache is the subject of the previous chapter, but in terms of Nginx configuration, you can see that it's fairly simply a matter of just pointing the configuration in the direction of the SSL certificate and key file for the website domain:

```
ssl on;
ssl_certificate     /etc/ssl/example.com/example.com.crt;
ssl_certificate_key /etc/ssl/example.com/example.com.key;
```

The NodeJS Application

If you want to see the test Node.js server's, they are as follows. You can use the following in a `server.js` file:

File: /srv/server.js

```
var http = require('http');

function serve(ip, port)
{
     http.createServer(function (req, res) {
        res.writeHead(200, {'Content-Type': 'text/plain'}); // Return a 200 resp\
onse
        res.write(JSON.stringify(req.headers));             // Respond with requ\
est headers
        res.end("\nServer Address: "+ip+":"+port+"\n");     // Let us know the s\
erver that responded
    }).listen(port, ip);
    console.log('Server running at http://'+ip+':'+port+'/');
}

serve('127.0.0.1', 9000);
serve('127.0.0.1', 9001);
serve('127.0.0.1', 9002);
```

This listens for HTTP requests on 3 sockets, simulating three web servers for the Nginx load balancer to use.

This "application" simply prints out the request headers received in the HTTP request, allowing us to inspect the headers and other information the load balancer sends.

Load Balancing with HAProxy

While there are quite a few good options for load balancers, HAProxy has become a go-to Open Source solution.

It's used by many large companies, including GitHub, Stack Overflow, Reddit, Tumblr and Twitter.

HAProxy (High Availability Proxy) is able to handle a lot of traffic. Similar to Nginx, it uses a single-process, event-driven model. This uses a low (and stable) amount of memory, enabling HAProxy to handle a large number of concurrent requests.

Setting it up is pretty easy as well! We'll cover installing and setting up HAProxy to load balance between three sample NodeJS HTTP servers., just like we did in the Nginx chapter.

Common Setups

In a typical (production) setup, web servers such as Apache or Nginx will stand between HAProxy and a web application. These web servers will typically either respond with static files or proxy requests they receive off to a Node, PHP, Ruby, Python, Go, Java or other dynamic application that might be in place.

Unlike Nginx, HAProxy can balance requests between any application that can handle HTTP or even TCP requests. In this example, setting up three NodeJS web servers is just a convenient way to show load balancing between three web servers. How HAProxy sends requests to a web server or TCP end point doesn't end up changing how HAProxy works!

If you've purchased the case studies as well, you can read one which covers TCP load balancing to distribute traffic amongst MySQL read-servers in a replica setup.

Installation

We'll install the latest HAProxy, 1.5.4 as of this writing. To do so, we can use the `ppa:vbernat/haproxy-1.5` repository, which will get us a recent stable release:

```
sudo add-apt-repository -y ppa:vbernat/haproxy-1.5
sudo apt-get update
sudo apt-get install -y haproxy
```

HAProxy Configuration

HAProxy configuration can be found at `/etc/haproxy/haproxy.cfg`. Here's what we'll likely see by default:

```
global
    log /dev/log    local0
    log /dev/log    local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
    user haproxy
    group haproxy
    daemon

    # Default SSL material locations
    ca-base /etc/ssl/certs
    crt-base /etc/ssl/private

    # Default ciphers to use on SSL-enabled listening sockets.
    # For more information, see ciphers(1SSL).
    ssl-default-bind-ciphers kEECDH+aRSA+AES:kRSA+AES:+AES256:RC4-SHA:!kEDH:!LOW\
:!EXP:!MD5:!aNULL:!eNULL

defaults
    log     global
    mode    http
    option  httplog
    option  dontlognull
    timeout connect 5000
    timeout client  50000
    timeout server  50000
    errorfile 400 /etc/haproxy/errors/400.http
    errorfile 403 /etc/haproxy/errors/403.http
    errorfile 408 /etc/haproxy/errors/408.http
    errorfile 500 /etc/haproxy/errors/500.http
    errorfile 502 /etc/haproxy/errors/502.http
```

```
33      errorfile 503 /etc/haproxy/errors/503.http
34      errorfile 504 /etc/haproxy/errors/504.http
```

Here we have some global configuration, and then some defaults (which we can override as needed for each `server` setup).

Within the `global` section, we likely won't need to make any changes. Here we see that HAProxy is run as the user/group `haproxy`, which is created during install. Running as a separate system user/group provides some extra avenues for increasing security through user/group permissions.

Furthermore, the master process is run as root - that process then uses `chroot` to separate HAProxy from other system areas, almost like running within its own container.

HAProxy also setes itself as running as a `daemon` (in the background).

The `log` directives don't actually log to specific files. Instead, HAProxy uses `rsyslog`, which is covered in the Log section of this book. This sends logs to the system logger, which is then responsible for routing logs to the appropriate place.

We'll cover HAProxy `stats` later, but this sets up some defaults for HAProxy to send statistics, useful for monitoring.

Within `defaults` section, we see some logging and timeout options. HAProxy can log all web requests, giving you the option to turn off access logs in each web node, or conversely, turning logs off at the load balancer while having them on within each web server (or any combination thereof). Where you want your logs to be generated/saved/aggregated is a decision you should make based on your needs.

If you want to turn off logging regular (successful) HTTP requests within HAProxy, add the line `option dontlog-normal`. The dontlog-normal directive[75] will tell HAProxy to only log error responses from the web nodes. Alternatively, you can simply separate error logs from the regular access logs via the `option log-separate-errors`[76] option.

Note that this puts HAProxy in `http` mode, which means it will operate as if received requests are HTTP requests. HAProxy can also handle TCP requests, in which case a mode of `tcp` will be used. These defaults can usually stay as they are, as they will be over-ridden as needed in the individual `server` sections.

Load Balancing Configuration

To get started balancing traffic between our three HTTP listeners, we need to set some options within HAProxy:

- `frontend` - where HAProxy listens for incoming connections

[75] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20dontlog-normal

[76] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4.2-option%20log-separate-errors

- `backend` - Where HAPoxy sends incoming connections to
- `stats` - Optionally, setup HAProxy web tool for monitoring the load balancer and its nodes

Here's an example **frontend**:

```
frontend localnodes
        bind *:80
        mode http
        default_backend nodes
```

This is a frontend, which I have arbitrarily named 'localnodes', since I'm running NodeJS processes locally.

- `bind *:80` - I've bound this frontend to all network interfaces on port 80. HAProxy will listen on port 80 on each available network for new HTTP connections
- `mode http` - This is listening for HTTP connections. HAProxy can handle lower-level TCP connections as well, which is useful for load balancing things like MySQL read databases if you setup database replication
- `default_backend nodes` - This frontend should use the backend named `nodes`, which we'll see next.

  TCP is "lower level" than HTTP. HTTP is actually built on top of TCP, so every HTTP connection is a TCP connection, but not every TCP connection is an HTTP request.

Next let's see an example **backend** configuration:

```
backend nodes
    mode http
    balance roundrobin
    option forwardfor
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    server web01 172.0.0.1:9000 check
    server web02 172.0.0.1:9001 check
    server web03 172.0.0.1:9002 check
```

This is where we configure the servers to distribute traffic between. I've named the backend "nodes". Similar to the frontend, the name is arbitrary. Let's go through the options seen there:

- `mode http` - This will pass HTTP requests to the servers listed

- `balance roundrobin` - Use the roundrobin[77] strategy for distributing load amongst the servers
- `option forwardfor` - Adds the `X-Forwarded-For` header so our applications can get the client's actual IP address. Without this, our application would instead see every incoming request as coming from the load balancer's IP address
- `http-request set-header X-Forwarded-Port %[dst_port]` - We manually add the `X-Forwarded-Port` header so that our applications knows what port to use when redirecting/-generating URLs.
  - Note that we use the `dst_port`[78] "destination port" variable, which is the destination port of the *client's* HTTP request, not of the upstream (NodeJS in this example) servers.
- `http-request add-header X-Forwarded-Proto https if { ssl_fc }` - We add the `X-Forwarded-Proto` header and set it to "https" if an SSL connection is used. Similar to the forwarded-port header, this can help our web applications determine which scheme to use when building URL's and sending redirects (Location headers).
- `option httpchk HEAD / HTTP/1.1\r\nHost:localhost` - Set the health check URI which HAProxy uses to test if the web servers are still responding. If these fail to respond, the server is removed from HAProxy as one to load balance between. This sends a HEAD request with the `HTTP/1.1` and `Host` header set, which might be needed if your web server uses virtualhosts to detect which site to send traffic to.
- `server web[01-03] 172.0.0.0:[9000-9002] check` - These three lines add the web servers for HAProxy to balance traffic between. It arbitrarily names each one `web01-web03`, then set's their IP address and port, and adds the `check` directive to tell HAProxy to health check the server as directed by `option httpchk`.

Load Balancing Algorithms

Let's take a quick minute to go over something important to load balancing - deciding how to distribute traffic amongst the upstream servers. The following are a few of the options HAProxy offers in version 1.5+:

Roundrobin: In the above configuration, we used the pretty basic `roundrobin` algorithm to distribute traffic amongst our three servers. With `roundrobin`, each server is used in turn (although you can add weights to each server). It is limited by design to "only" 4095 servers.

Weights[79] default to 1, and can be as high as 256. Since we didn't set one above, all have a weight of 1, and roundrobin simply goes from one server to the next.

We can use **sticky sessions** with this algorithms. Sticky sessions are user sessions, usually identified by a cookie, which helps HAProxy to always send requests to the same server for a particular client.

[77] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4.2-balance
[78] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#7.3.3-dst_port
[79] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#weight

This is useful for web applications that use default session handling, which likely saves session data on the server, rather than within a browser cookie or in a centralized session store such as redis or memcached.

In that scenario, users must be returned to the same server on which they created their session ("logged in") in order to remain in that session.

To use sticky sessions, you can add a `cookie SOME-COOKIE-NAME prefix` directive into the `backend` section. Then simply add the `cookie` parameter within each server. HAProxy will then append a new cookie identifier for each server. This cookie will be sent back in subsequent requests from the client, letting HAProxy know which server to send the request to. This looks like the following:

```
backend nodes
    # Other options above omitted for brevity
    cookie SRV_ID prefix
    server web01 172.0.0.1:9000 check cookie web01
    server web02 172.0.0.1:9001 check cookie web02
    server web03 172.0.0.1:9002 check cookie web03
```

I suggest using cookie-based sessions or a central session store rather than default server sessions if you have the option to do so within your web applications. Don't rely on requiring clients to always connect to the same web server to stay logged into your application, as the mechanism can fail if cookies are modified in the browser and this setup can distribute traffic unevenly.

static-rr: This is similar to the round-robin method, except you can't adjust server weights on the fly. In return, it has no design limitation on the number of servers, like standard round-robin does.

leastconn: The server with the lowest number of connections receives the connection. This is better for servers with long-running connections (LDAP, SQL, TSE, web sockets, long polling), but not necessarily for short-lived connections (regular HTTP).

uri: This takes a set portion of the URI used in a request, hashes it, divides it by the total weights of the running servers and uses the result to decide which server to send traffic to. This effectively makes it so the same server handles the same URI end points.

This is often used with proxy caches in order to maximize the cache hit rate.

Not mentioned, but worth checking out in the documentation are the remaining balancing algorithms:

- rdp-cookie - Session stickiness for the RDP protocol
- first
- source
- url_param
- hdr

Test the Load Balancer

Putting all those directives inside of the `/etc/haproxy/haproxy.cfg` file gives us a load balancer!

Here's the complete configuration file at `/etc/haproxy/haproxy.cfg`:

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/haproxy
        stats socket /run/haproxy/admin.sock mode 660 level admin
        stats timeout 30s
        user haproxy
        group haproxy
        daemon

        # Default SSL material locations
        ca-base /etc/ssl/certs
        crt-base /etc/ssl/private

        # Default ciphers to use on SSL-enabled listening sockets.
        # For more information, see ciphers(1SSL).
        ssl-default-bind-ciphers kEECDH+aRSA+AES:kRSA+AES:+AES256:RC4-SHA:!kEDH:\
!LOW:!EXP:!MD5:!aNULL:!eNULL

defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
        errorfile 400 /etc/haproxy/errors/400.http
        errorfile 403 /etc/haproxy/errors/403.http
        errorfile 408 /etc/haproxy/errors/408.http
        errorfile 500 /etc/haproxy/errors/500.http
        errorfile 502 /etc/haproxy/errors/502.http
        errorfile 503 /etc/haproxy/errors/503.http
        errorfile 504 /etc/haproxy/errors/504.http

frontend localnodes
    bind *:80
```

```
    mode http
    default_backend nodes

backend nodes
    mode http
    balance roundrobin
    option forwardfor
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    server web01 172.0.0.1:9000 check
    server web02 172.0.0.1:9001 check
    server web03 172.0.0.1:9002 check

listen stats *:1936
    stats enable
    stats uri /
    stats hide-version
    stats auth someuser:password
```

You start/restart/reload start HAProxy with these settings. Below I restart HAProxy just because if you've been following line by line, you may not have started HAProxy yet:

```
# You can reload if HAProxy is already started
$ sudo service haproxy restart
```

Then start the Node server:

```
# Example node server seen below
node /srv/server.js
```

I'm assuming the Node server is being run on the same server has HAProxy for this example - that's why all the IP addresses used are referencing localhost `127.0.0.1`.

Then head to your server's IP address or hostname and see it balance traffic between the three Node servers. I broke out the first request's headers a bit so we can see the added `X-Forwarded-*` headers:

```
{"host":"192.169.22.10",
"cache-control":"max-age=0",
"accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q\
=0.8",
"user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 \
(KHTML, like Gecko) Chrome/35.0.1916.153 Safari/537.36",
"accept-encoding":"gzip,deflate,
sdch","accept-language":"en-US,en;q=0.8",
"x-forwarded-port":"80",             // Look, our x-forwarded-port header!
"x-forwarded-for":"172.17.42.1"}     // Look, our x-forwarded-for header!
There's no place like 0.0.0.0:9000   // Our first server, on port 9000

{"host":"192.169.22.10", ... }
There's no place like 0.0.0.0:9001  // Our second server, on port 9001

{"host":"192.169.22.10", ... }
There's no place like 0.0.0.0:9002  // Our third server, on port 9002
```

See how it round-robins between the three servers in the order they are defined! We also have the `x-forwarded-for` and `x-forwarded-port` headers available to us, which our application can use

Monitoring HAProxy

You may have noticed the following directives which I haven't discussed yet:

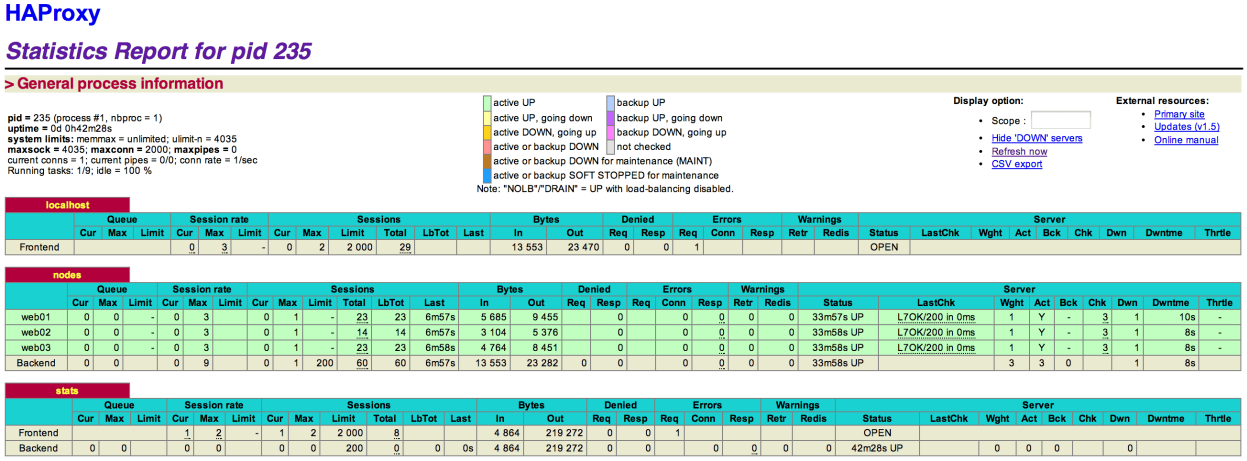
```
listen stats *:1936
    stats enable
    stats uri /
    stats hide-version
    stats auth someuser:password
```

HAProxy comes with a web interface for monitoring the load balancer and the upstream server statuses. Let's go over the above options:

- `listen stats *:1936` - Use the `listen` directive, name it `stats` and have it listen on port `1936` for all network interfaces.
- `stats enable` - Enable the stats monitoring dashboard
- `stats uri /` - The URI to reach it is just `/` (on port 1936)
- `stats hide-version` - Hide the version of HAProxy used

- `stats auth someuser:password` - This uses HTTP basic authentication, with the set username and password. In this example, the username is `someuser` and the password is just `password`. Don't use those in production - in fact, make sure your firewall blocks public, external HTTP access to your configured port

When you head to your server and port in your web browser, here's what the dashboard will look like:



HAProxy

Statistics Report for pid 235

> General process information

pid = 235 (process #1, nbproc = 1)
uptime = 0d 0h42m28s
system limits: memmax = unlimited; ulimit-n = 4035
maxsock = 4035; maxconn = 2000; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec
Running tasks: 1/9; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

localhost

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	3	-	0	2	2 000	29			13 553	23 470	0	0	1					OPEN								

nodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
web01	0	0	-	0	3		0	1	-	23	23	6m57s	5 685	9 455		0		0	0	0	0	33m57s UP	L7OK/200 in 0ms	1	Y	-	3	1	10s	-
web02	0	0	-	0	3		0	1	-	14	14	6m57s	3 104	5 376		0		0	0	0	0	33m58s UP	L7OK/200 in 0ms	1	Y	-	3	1	8s	-
web03	0	0	-	0	3		0	1	-	23	23	6m58s	4 764	8 451		0		0	0	0	0	33m58s UP	L7OK/200 in 0ms	1	Y	-	3	1	8s	-
Backend	0	0		0	9		0	1	200	60	60	6m57s	13 553	23 282	0	0		0	0	0	0	33m58s UP		3	3	0		1	8s	

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	2	-	1	2	2 000	8			4 864	219 272	0	0	1					OPEN								
Backend	0	0		0	0		0	0	200	0	0	0s	4 864	219 272	0	0		0	0	0	0	42m28s UP		0	0	0		0		

HAProxy Stats

The IP address `192.168.22.10` happened to be the IP address of my test server.

We can see the Frontend we defined under `localhost` (actually named `localnodes` in the configuration above). This section shows the status of incoming requests.

There is also the `nodes` section (the name I chose for the defined backend section), our defined backend servers. Each server here is green, which shows that they are "healthy". If a health check fails on any of the three servers, then it will display as red and it won't be included in the rotation of the servers.

Finally there is the stats section, which just gives information about the stats page that shows this very information.

Sample NodeJS Web Server

To keep this example simple, we've use a previously mentioned NodeJS application, which just opens up three HTTP listeners on separate ports:

File: /srv/server.js

```
var http = require('http');

function serve(ip, port)
{
        http.createServer(function (req, res) {
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.write(JSON.stringify(req.headers));
            res.end("\nThere's no place like "+ip+":"+port+"\n");
        }).listen(port, ip);
        console.log('Server running at http://'+ip+':'+port+'/');
}

// Create three servers for
// the load balancer, listening on any
// network on the following three ports
serve('0.0.0.0', 9000);
serve('0.0.0.0', 9001);
serve('0.0.0.0', 9002);
```

We bounced traffic between these three web servers with HAProxy. This “application” simply responds to any HTTP request with the IP address/port of the server, along with the request headers received in the HTTP request.

SSL with HAProxy

If your application makes use of SSL certificates, then some decisions need to be made about how to use them with the a load balancer.

A simple setup of **one** web server usually sees a client's SSL connection being unencrypted by the server receiving the request. However, a load balancer will usually be a "gateway" into an application - it is a central point into which all (most) traffic goes, before being distributed to upstream servers. Because of this, where an SSL connection is unencrypted becomes a concern.

As previously discussed, there are a few strategies for handling SSL connections with load balancers:

SSL Termination is the practice of terminating/decrypting an SSL connection at the load balancer, and sending unencrypted connections to the backend servers.

This is the opposite of **SSL Pass-Through**, which sends SSL connections directly to the proxied servers. The SSL connection is terminated at each proxied server.

This means your application servers will lose the ability to get the `X-Forwarded-*` headers, which may include the client's IP address, port and scheme used.

Lastly, you can use a combination of **both** strategies, where SSL connections are terminated at the load balancer, adjusted as needed, and then proxied off to the backend servers as a *new* SSL connection.

Which strategy you choose is up to you and your application needs. SSL Termination is the most typical.

HAProxy with SSL Termination

We'll cover the most typical use case first - **SSL Termination**. As stated, we need to have the load balancer handle the SSL connection. This means having the SSL Certificate live on the load balancer server.

We saw how to create an SSL certificate in a previous chapter. We'll re-use that information for setting up a self-signed SSL certificate for HAProxy to use.

Keep in mind that for a production SSL Certificate (not a self-signed one), you won't need to generate or sign the certificate yourself - you'll just need to create a Certificate Signing Request (CSR) and pass that to whomever you purchase a certificate from. After a verification process, you'll receive a valid SSL certificate which you can install in the same way we'll do in this chapter.

In this chapter, we'll create a self-signed certificate for `*.xip.io`, which is handy for demonstration purposes, and lets use one the same certificate when our server IP addresses might change while testing locally. For example, if our local server exists at 192.168.33.10, but our server's IP changes to 192.168.33.11, then we don't need to re-create the self-signed certificate.

> I use the `xip.io` service as it allows us to use a hostname rather than directly accessing the servers via an IP address, all without having to edit my computers' Host file. See chapter "DNS & Hosts File" for more information.

The process of creating an SSL certificate is covered in a previous chapter, so I'll just show the commands to create the self-signed SSL certificate:

```
$ sudo mkdir /etc/ssl/xip.io
$ sudo openssl genrsa -out /etc/ssl/xip.io/xip.io.key 2048
$ sudo openssl req -new -key /etc/ssl/xip.io/xip.io.key \
                   -out /etc/ssl/xip.io/xip.io.csr

Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Connecticut
Locality Name (eg, city) []:New Haven
Organization Name (eg, company) [Internet Widgets Pty Ltd]:SFH
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:*.xip.io
Email Address []:

Please enter the following 'extra' attributes to be sent with your certificate r\
equest
A challenge password []:
An optional company name []:
$ sudo openssl x509 -req -days 365 -in /etc/ssl/xip.io/xip.io.csr \
                    -signkey /etc/ssl/xip.io/xip.io.key \
                    -out /etc/ssl/xip.io/xip.io.crt
```

This leaves us with a `xip.io.csr`, `xip.io.key` and `xip.io.crt` files in the `/etc/ssl/sfh` directory.

If you're purchasing an SSL certificate, skip the last step, as you'll receive certificate files from the registrar of your purchased SSL certificate.

After the certificates are created, we need to create a `.pem` file. A `.pem` file is essentially just the certificate, the key and optionally the root and any intermediary certificate authorities concatenated into one file. Because we are generating a self-signed certificate, there are no certificate authorities

in play, and so we'll simply concatenate the certificate and key files together (*in that order*) to create a `xip.io.pem` file.

Using one concatenated file for the SSL certificate information is HAProxy's preferred way to read an SSL certificate:

```
$ sudo cat /etc/ssl/xip.io/xip.io.crt /etc/ssl/xip.io/xip.io.key \
          > /etc/ssl/xip.io/xip.io.pem
```

When purchasing a real certificate, you might get a concatenated "bundle" file. If you do, it might not be a `pem` file, but instead be a `bundle`, `cert`, `cert`, `key` file or some similar name for the same concept. You'll need t inspect the files or follow instructions provided for you to find out which you receive.

If you do not receive a bundled file, you may have to concatenate them yourself in the order of certificate, private key, any intermediaries certificate authority (CA) certificates and lastly the root CA certificate.

This Stack Overflow answer[80] explains some certificate file-formats nicely.

In any case, once we have a `.pem` file for HAproxy to use, we can adjust our configuration just a bit to handle SSL connections.

We'll setup our application to accept both `http` and `https` connections. In the previous section, we defined this frontend:

File: /etc/haproxy/haproxy.cfg

```
frontend localnodes
    bind *:80
    mode http
    default_backend nodes
```

To terminate an SSL connection in HAProxy, we can now add a binding to the standard SSL port 443, and let HAProxy know where the SSL certificates are:

```
frontend localhost
    bind *:80
    bind *:443 ssl crt /etc/ssl/xip.io/xip.io.pem
    mode http
    default_backend nodes
```

In the above example, we're using the backend "nodes". The backend, luckily, doesn't really need to be configured in any particular way. We can configure a backend as we normally would:

[80] http://serverfault.com/questions/9708/what-is-a-pem-file-and-how-does-it-differ-from-other-openssl-generated-key-file

```
backend nodes
    mode http
    balance roundrobin
    option forwardfor
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }
    server web01 172.17.0.3:9000 check
    server web02 172.17.0.3:9001 check
    server web03 172.17.0.3:9002 check
```

Because the SSL connection is terminated at the Load Balancer, we're still sending regular HTTP requests to the backend servers. We don't need to change this configuration, as it works the same!

SSL Only

If you'd like the site to be SSL-only, you can add a `redirect` directive to the frontend configuration:

```
frontend localhost
    bind *:80
    bind *:443 ssl crt /etc/ssl/xip.io/xip.io.pem
    redirect scheme https if !{ ssl_fc }
    mode http
    default_backend nodes
```

Above, we added the `redirect` directive, which will redirect from "http" to "https" if the connection was not made with an SSL connection. More information on `ssl_fc` is available in the documentation[81].

HAProxy with SSL Pass-Through

With **SSL Pass-Through**, we'll have our backend servers handle the SSL connection, rather than the load balancer.

The job of the load balancer then is simply to proxy a request off to its configured backend servers. Because the connection remains encrypted, HAProxy can't do anything with it other than redirect a request to another server.

In this setup, we need to use TCP mode over HTTP mode in both the frontend and backend configurations. HAProxy will treat the connection as just a stream of information to proxy to a server, rather than use its functions available for HTTP requests.

First, we'll tweak the `frontend` configuration:

[81] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#7.3.4-ssl_fc

```
frontend localhost
    bind *:80
    bind *:443
    option tcplog
    mode tcp
    default_backend nodes
```

This still binds to both port 80 and port 443, giving the opportunity to use both regular and SSL connections.

As mentioned, to pass a secure connection off to a backend server without encrypting it, we need to use TCP mode (`mode tcp`) instead. This also means we need to set the logging to tcp instead of the default http (via `option tcplog`). There is more information about the difference between `tcplog` and `httplog` log formats in the documentation[82].

Next we need to tweak our `backend` configuration. we once again need to change this to TCP mode, and we remove some directives to reflect the loss of ability to edit/add HTTP headers:

```
backend nodes
    mode tcp
    balance roundrobin
    option ssl-hello-chk
    server web01 127.0.0.1:9000 check
    server web02 127.0.0.1:9001 check
    server web02 127.0.0.1:9002 check
```

As you can see, this is set to `mode tcp` - Both frontend and backend configurations need to be set to this mode.

We also remove `option forwardfor` and the `http-request` options - these can't be used in TCP mode as we're not reading it as an HTTP request. We can't read or inject headers into a request that's encrypted.

Keep in mind the pitfalls of using SSL Pass-Thru as discussed in previous chapters. Your application may require the actual client information (IP address) for certain functionality, but instead receive the load balancer's information when using SSL Pass-Thru. We can't inject the `X-Forwarded-*` headers using this method, so any client-specific logging and functionality (perhaps throttling) would need to be done at the load-balancer.

For health checks, we can use `ssl-hello-chk` which checks the connection as well as its ability to handle SSL (SSLv3 specifically by default) connections.

[82] http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#8.2

In this example, I have three fictitious server backend that accept SSL certificates. If you've read the chapter "SSL Certificates", you can see how to integrate them with Apache or Nginx in order to create a web server backend which handles SSL traffic. In this example, I use a NodeJS server, which listens for SSL traffic on ports 9000-9002.

With SSL Pass-Through, no SSL certificates need to be created or used within HAProxy. The backend servers can handle SSL connections just as they would if there was only one server used in the stack without a load balancer.

Sample NodeJS Web Server

If you want to test SSL Pass-Thru with a sample NodeJS server, we can do that as well. I tested the SSL Pass-Thru section with the following NodeJS server, and ran it on the same server as HaProxy. Note that you don't need to listen on port 443 to use SSL certificates:

File: /srv/server.js

```
var https = require('https');
var fs = require('fs');

var options = {
    key: fs.readFileSync('/etc/ssl/xip.io/xip.io.key'),
    cert: fs.readFileSync('/etc/ssl/xip.io/xip.io.crt')
};

function serve(ip, port)
{
        https.createServer(options, function (req, res) {
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.write(JSON.stringify(req.headers));
            res.end("\nThere's no place like "+ip+":"+port+"\n");
        }).listen(port, ip);
        console.log('Server running at http://'+ip+':'+port+'/');
}

// Create three servers for
// the load balancer, listening on any
// network on the following three ports
serve('0.0.0.0', 9000);
serve('0.0.0.0', 9001);
serve('0.0.0.0', 9002);
```

We bounced traffic between these three web servers with HAProxy. This "application" simply responds to any HTTPS request with the IP address/port of the server, along with the request headers received in the HTTPS request.

Web Cache

A large portion of the HTTP protocol RFC2616[83] defines various cache mechanisms available to us. Knowing these mechanisms and how to implement them is a valuable asset to web programming.

If you are not familiar with HTTP cache mechanisms, Mark Nottingham has written a great primer on web caching[84].

In the following chapters, we'll briefly cover some caching "theory". Then we'll concentrate on how to make use of HTTP caching in Nginx and Varnish.

[83] http://www.w3.org/Protocols/rfc2616/rfc2616.html
[84] https://www.mnot.net/cache_docs/

Nuts and Bolts of HTTP Caching

In the world of web application development, we generally talk about two types of caches:

1. Object (in-memory) Cache
2. Web (HTTP) Cache

Object Caches

Applications use an object cache to store the result of expensive operations. Object caches can store the results of slow database queries or expensive calculations.

Two of the most popular object caches are Memcached and Redis.

We will not be discussing object caches in this section.

Web Caches

Web caching is built into the HTTP protocol. Any HTTP client and server can implement all or part of the HTTP caching specification.

Interestingly, HTTP clients (your browser, the curl cli command, code consuming an API) are responsible for a good portion of caching.

The clients are responsible for implementing caching rules set by an origin server. The origin server is responsible for setting the rules.

There are three main types of web caches:

- **Proxy** - A proxy cache is a public, shared cache often employed by an ISP or large corporation. Because they are employed at a high-level, they can (and do) cache thousands of various websites.
- **Gateway** (reverse-proxy) - Similar to in-app caches, Gateway caches are part of your infrastructure. They sit in front of your web-server and act much in the same way of proxy cache. Private caches, however, are for your application(s) only. Similar to a load balancer, they are technically a reverse proxy. Varnish is a gateway cache.
- **Private** - Private caches are unique to a specific user. They live on the client-end. Your browser is a private cache; it will cache responses unique to the sites you visit based on the rules set by the origin server.

In this book, we'll be discussing the installation and use of Gateway caches. These are also called reverse-proxy caches.

A `resource` is a URI. The URI may point to a file, a directory, or to a dynamic application. Any URI, with or without query parameters, is a resource.

An `origin server` is the server which contains the resource requested. These may be images, css files, javascript, or the results of dynamic requests.

The origin server typically defines how the resource is to be cached via HTTP response headers.

Types of HTTP Caches

The HTTP specification defines 2 methods of HTTP caching.

Validation Caching

Validation caching requires that clients validate if a resource has changed before serving it. This potentially saves bandwidth, but still requires an HTTP request be sent all the way to the origin server.

> Validation Caching is often used by our browsers without us meaning to. ETags or similar headers may be turned on for files handled by your web server after a default installation with no configuration. We web developers often want to use Expiration Caching instead.

They do this by sending an HTTP request to the origin server asking if the resource was modified since it last checked.

If the resource has not changed, the origin server does not need to reply with a full message body (with the full resource). A bodyless HTTP response with only headers is sent back. This let's the client know to use the last version of the resource downloaded. This of course assumes the client has saved the resource the last time it received it.

Accomplished with `if-*` headers (`If-None-Match`, `If-Modified-Since`, etc), validation caching is primarily used for 2 things:

1. **Conditional GET** - A server can tell the request 'nothing has changed since you last checked'. This is great for mobile APIs where the bandwidth of re-sending a message body can be expensive.

2. **Concurrency Control** - In a POST or PUT request, a server can check if the resource was changed since the requester last checked. This helps solve the "Lost Update Problem" where the last write may overwrite a previous write to a resource. Think of two people opening the same Word document. The last person to hit save will overwrite the previous person. This is good for APIs with a lot of writes (updates) to resources.

With validation caching, a request is always sent to the origin server. The origin server can determine if the resource has changed and either send a full response or a `304 Not Modified` response in return.

This saves in bandwidth rather than server processing.

Expiration Caching

This is the caching we're most familiar with. Expiration headers tell HTTP clients how long a resource (file) can be considered "fresh".

After this time period, a resource is considered "stale". The client keeps track of this and therefore knows it must re-download the resource from the origin server. The resource can be re-cached for the time defined by the returned HTTP headers.

Rather than just bandwidth, this saves any HTTP requests from having to be made to the origin server. A cache (or client private cache) can serve a response directly. This saves the origin server from processing extra requests.

Expiration rules are set with the `Age`, `Expires`, `Cache-Control`, `Date` and related headers.

An Origin Server

In the remaining chapters, we'll see how to install and configure HTTP cache servers.

Before we do, let's explore setting up an origin server to properly serve static assets. An origin server can be any old web server.

We'll use Nginx to to see how to configure a web server to support validation and expiration cache mechanisms.

Cache servers sit between a client and the origin servers. Origin servers are responsible for setting cache policy. They return validation and expiration cache information via HTTP response headers.

Cache servers will use this policy to determine when and how to cache files.

For validation caching, the cache server will send a request to the origin server to check if the resource has changed. If it has not, the cache server will respond with its cached copy of the resource.

For expiration caching, the cache server will store files for as long as it's permitted. It will respond with its cached version of a resource until that resource becomes stale (expires).

Testing Caching Mechanisms

We'll test out our origin server which, as stated, happens to be Nginx. Note that the following examples are showing caching mechanisms between a client (curl requests I'll make on the command line) and the origin server (Nginx).

Our goal is to configure the origin server, which must be configured properly before we add in a proxy cache such as Varnish.

We'll send requests for a CSS file as an example.

Validation Caching

Valiation caching mechanisms are easy to test with a tool such as `curl`.

Here we'll request a CSS file and view the response.

Making requests on fresh install of Nginx - no extra configuration

```
$ curl -I  http://localhost/static/css/styles.css
HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Fri, 24 Oct 2014 00:30:59 GMT
Content-Type: text/css
Content-Length: 11767
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
Connection: keep-alive
ETag: "52b9b863-2df7"
Accept-Ranges: bytes
```

The response contains an ETag. ETags are generated based on the contents of a file. **If a file changes, the ETag returned by the origin server will be changed**.

Let's play with Validation Caching mechanisms and see if Nginx implements them.

Using Etags and If-None-Match to test Validation Caching

```
$ curl -I \
    -H 'If-None-Match: "52b9b863-2df7"' \
    http://localhost/static/css/styles.css
HTTP/1.1 304 Not Modified
Server: nginx/1.6.2
Date: Fri, 24 Oct 2014 00:31:44 GMT
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
Connection: keep-alive
ETag: "52b9b863-2df7"
```

We asked Nginx to return the CSS file if it is not modified. We supplied an ETag, which is the string given to use when we first requested the CSS file.

Since the ETag we gave still matches the one Nginx generates for the file, Nginx can say that the requested CSS file has not been modified since the last time the client asked for it.

We can see that Nginx responded with a `304 Not Modified` response as expected!

Next we'll test date-based Validation Caching:

Validation caching with dates over ETags

```
$ curl -I \
    -H 'If-Modified-Since: Tue, 24 Dec 2013 16:37:55 GMT' \
    http://localhost/static/css/styles.css
HTTP/1.1 304 Not Modified
Server: nginx/1.6.2
Date: Fri, 24 Oct 2014 00:37:06 GMT
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
Connection: keep-alive
ETag: "52b9b863-2df7"
```

Rather than use an ETag, we asked if the file has been modified since a given date. Once again, we get a `304 Not Modified` response. The file was not modified since the given date!

Expiration Caching

We can't test Expiration Caching in exactly the same way, as it's up to the client to handle the expiration cache policy. The origin server only gives resource expiration information.

We can, however, ensure that the origin server is sending us proper cache headers.

By default, Nginx **won't** set cache expiration headers. This is something we need to define ourselves. I take the following expiration configuration from H5BP's Nginx Server Configuration[85] repository:

Adding cache headers for static assets in Nginx configuration

```
# cache.appcache, your document html and data
location ~* \.(?:manifest|appcache|html?|xml|json)$ {
  expires -1;
}

# Feed
location ~* \.(?:rss|atom)$ {
  expires 1h;
  add_header Cache-Control "public";
}

# Media: images, icons, video, audio, HTC
location ~* \.(?:jpg|jpeg|gif|png|ico|cur|gz|svg|svgz|mp4|ogg|ogv|webm|htc)$ {
  expires 1M;
  access_log off;
```

[85] https://github.com/h5bp/server-configs-nginx

```
16    add_header Cache-Control "public";
17  }
18
19  # CSS and Javascript
20  location ~* \.(?:css|js)$ {
21    expires 1y;
22    access_log off;
23    add_header Cache-Control "public";
24  }
```

You can add the above to your Nginx site configuration.

Note that we're telling the server that files ending in `.xml`, `.json`, `.html` and other extensions aren't to be cached. You're free to change this for your needs.

Other files have their `cache-control` set to `public`, effectively meaning any cache is allowed to store the file. The expiration time is then set separately per resource type. Some files expire in one month, while others expires in one year.

> H5BP tends to cache static assets for a long time, assuming you'll use some kind of cache busting mechanism, such as appending a query string at the end of the file. For example `styles.css?1234567890` is cached as a separate file from `styles.css?09876543221`.

Once the configuration is added and we reload Nginx, we can test the responses we get:

Testing for expiration cache headers

```
1  $ curl -I  http://localhost/static/css/styles.css
2  HTTP/1.1 200 OK
3  Server: nginx/1.6.2
4  Date: Fri, 24 Oct 2014 01:13:43 GMT
5  Content-Type: text/css
6  Content-Length: 11767
7  Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
8  Connection: keep-alive
9  ETag: "52b9b863-2df7"
10 Expires: Sat, 24 Oct 2015 01:13:43 GMT
11 Cache-Control: max-age=31536000
12 Cache-Control: public
13 Accept-Ranges: bytes
```

Now we have cache expiration headers along with our Etag. Our `Cache-Control` header informs us that this file can be cached by public caches, and it has a max-age of 31536000 seconds (1 year).

Now that we have both cache methods enabled, our web server is setup enough to use with HTTP cache servers.

Modern browsers will prefer expiration caching. Validation caching is typically used if no expiration information is present. If you wish to have clients use validation caching mechanisms, the `Cache-Control: no-cache` header should be set and the `Etag` or `Last-Modified` headers should be present in responses.

Nginx Web Caching

Nginx is a very capable web cache. There are two things to know about Nginx:

- Nginx can serve static content (directly) very, very efficiently
- Nginx can also act as a "true" cache server when placed in front of application servers, just like you might with another cache server, such as Varnish

Many administrators reach for Varnish before it's really needed. While Varnish is a pure web cache with more cache features, Nginx may still be a perfect match for you.

If your traffic warrants adding a layer of infrastructure for caching, but not the overhead of introducing new technologies, Nginx might be a better fit.

This is especially true if you happen to use Nginx Plus, which comes with support and extra features.

Use Cases

Nginx handles static content well on it's own. This is a typical use case of a web server, rather than a cache server. However, since Nginx can proxy requests to other web servers or to applications (via HTTP, FastCGI and uWSGI), it's commonly used to increase performance for serving static files while proxying application requests to other processes.

Nginx can act as a static file server, a reverse proxy for web application and a load balancer. In addition to this, it can act as an HTTP cache server.

A benefit of Nginx is that these features are not necessarily mutually exclusive.

The following use-cases are popular for Nginx caching:

- Nginx can act as a "classic" HTTP cache server, sitting in front of another web server. It can intercept all requests, and then pass requests to the origin server if needed.
  - Nginx caching can be used in conjunction with a load balancer. In other words, Nginx can act as the load balancer and cache server all at once.
- Nginx can *also* cache the results of requests proxied to FastCGI and uWSGI processes! This usually means caching the results of application requests. A good use case is to cache the results of requests made to a CMS. The frontend seen by the public is typically "static" while the administrative area is typically dynamic.

In the example here, we'll put an Nginx cache server in front of another nginx web server. To be clear: there will be two instances of Nginx running, rather than one instance performing double duty.

How It Will Work

Origin Servers are the servers that have the actual static files or dynamically generated HTML. They have two responsibilities:

- Serve the dynamic and static content when requested
- Decide how files (and potentially dynamic content) should be cached, via the HTTP cache headers

Conversely, the Cache Server is the "frontman". It receives the intial HTTP request from a client. It then either returns a cached copy of the resource or passes the request off to the Origin server.

If the request is sent to the origin server, the origin's resonse headers are read by the cache server to determine if the response should be cached or simply passed through.

Some larger web applications use load balancers in addition to cache servers, resulting in a highly layered infrastructure. For example, a load balancer may get the original request from a client, then pass it to the web cache, where it may or may not finally reach an origin server.

Responsibilities of the Cache Server:

- Determine if HTTP request will accept a cached response
- Determine if there is a fresh item in the cache to respond with
- Send HTTP request to the origin server if a stale resource is requested
- Respond to a request from its cache or from the origin server as approprtiate

Our last actor here is the **Client**. Clients can have their own local (private) cache.

A client which implements a local cache has the following responsibilities:

- Sending requests
- Caching responses
- Deciding to pull requests from local cache or making HTTP request to retrieve them

Origin Server

The origin server is ultimately responsible for serving files and controlling how files are to be cached.

In the previous chapter, we covered setting up Nginx to act as an origin server. We'll pretend this server is up an running, listening on port `9000`. Its configuration might look something like this:

```
server {
    listen 9000 default_server;
    root /var/www/;
    index index.html index.htm;

    server_name example.com www.example.com;

    charset utf-8;

    # Include H5BP cache expiration
    # headers for static assets
    include h5bp/basic.conf;

    location / {
        try_files $uri $uri/ @proxy;
    }

    location @proxy {
        include proxy_params;
        proxy_pass http://127.0.0.1:8080;
    }
}
```

This origin server will serve static files and proxy to a web application listening on port `8080`.

Next we'll setup another instance of Nginx to act as a cache server. This cache server will receive requests before the origin server. It will decide whether or not to pass them off to the origin server.

Cache Server

The origin server is setup, but we need to implement a cache server to sit "in front of" the origin server.

In the scenario here, the Cache Server will be the web server receiving requests. It will pass HTTP requests off to the origin server if caching rules dictate so.

The following Nginx configuration is not implementing any caching yet. It will simply proxy requests to the origin server:

```
server {
    # Note that it's listening on port 80
    listen 80 default_server;

    server_name example.com www.example.com;

    charset utf-8;

    location / {
        include proxy_params;
        proxy_pass http://172.17.0.18:9000;
    }
}
```

This will listen for requests on port `80`, and proxy all of them to the origin server listening on port `9000`.

> The Cache server I setup for testing is listening on `172.17.0.13:80`. The origin server as listening on `172.17.0.18:9000`.

If we make a request on our cache server, we'll see exactly what we'd see as if we hit the origin server itself. This is because the cache server is currently not caching. It's just passing requests through:

```
$ curl -X GET -I 172.17.0.13/css/style.css
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Fri, 05 Sep 2014 23:30:07 GMT
Content-Type: text/css
Last-Modified: Fri, 05 Sep 2014 22:46:39 GMT
Expires: Sat, 05 Sep 2015 23:30:07 GMT
Cache-Control: max-age=31536000
Cache-Control: public
```

Now let's add the necessary items to have Nginx cache responses from the origin server. The following is the same virtual host we saw above, defined on the cache server. However I've added the cache directives needed by Nginx to act as a cache server:

```
proxy_cache_path /tmp/nginx levels=1:2 keys_zone=my_zone:10m inactive=60m;
proxy_cache_key "$scheme$request_method$host$request_uri";

server {
    listen 80 default_server;

    server_name example.com www.example.com;

    charset utf-8;

    location / {
        proxy_cache my_zone;
        add_header X-Proxy-Cache $upstream_cache_status;

        include proxy_params;
        proxy_pass http://172.17.0.18:9000;
    }
}
```

Let's cover the new items here.

proxy_cache_path

This first sets the path to *where* cache files are saved. The `/tmp` is usually world-writable, so that makes for an obvious location. The more security concious may create and specify a location that only Nginx can read and write to, however. In Debian/Ubuntu, this involves making a directory writable by user or group `www-data`.

The `levels` directive sets *how* cache files are saved to the file system. If this is not defined, cache file are saved directly in the path defined. If it is defined as such (`1:2`), cache files are saved in sub-directories of the cache path based on their md5 hashes.

The `keys_zone` is simply an arbitrary name of the "zone" which we refer to for this cache. Here it's named `my_zone` and is given 10MB of storage for cache keys and other meta data.

Note that this doesn't limit the amount of files that can be cached! It's just for meta data. The documentation claims that a 1MB zone can store ~8000 keys and meta data.

Finally we set the `inactive` directive. This tells Nginx to clear the cache of any asset that's not accessed within 60 minutes. The `inactive` directive defaults to 10 minutes if it is not explicitly set.

Note that `60m` for `inactive` is 60 minutes, while `10m` for `keys_zone` is 10 megabytes.

Using `inactive` gives Nginx the opportunity to "forget" about cached assets which are not commonly requested. This way Nginx caching gives the most bang for your buck. The most requested resources stay in the cache and follow HTTP cache rules as directed by the Origin Server.

proxy_cache_key

This is the key we use to differentiate cached files. The default is `$scheme$proxy_host$uri$is_args$args`, but we can change it if needed.

This can be set to something like `"$host$request_uri $cookie_user"` (with quotes) as well to incorporate cookies.

Cookies do affect caching, so be careful of your treatment with them! You may accidentally end up with a file being cached in duplicate per indivisual cookie (effectively per site visitor).

This means that incorporating cookies into the cache key does reduce the effectiveness of the cache. A cache per user is the purpose of a private cache (a web browser) rather than the "public" cache server we're building here. However there may be use cases in which you want to incorporate cookies. The option is available to you.

proxy_cache

Inside of the `location` block, we're telling Nginx to use the cache zone defined via the `proxy_cache my_zone` directive.

We also add a useful header which will inform us if the resource was served from cache or not. This is done via the `add_header X-Proxy-Cache $upstream_cache_status` directive. This sets a response header named `X-Proxy-Cache` with a value of either `HIT`, `MISS`, or `BYPASS`

Testing the Configuration

Once this is saved, we can reload the Nginx's configuration (`sudo service nginx reload`) and try some HTTP requests.

Here we attempt to get a CSS file for the first time:

```
# GET curl request and selected headers
$ curl -X GET -I 172.17.0.13/css/style.css
Date: Fri, 05 Sep 2014 23:50:12 GMT
Content-Type: text/css
Last-Modified: Fri, 05 Sep 2014 22:46:39 GMT
Expires: Sat, 05 Sep 2015 23:50:12 GMT
Cache-Control: max-age=31536000
Cache-Control: public
X-Proxy-Cache: MISS
```

This is a cache `MISS` because the file has not been requested before. Therefore the Cache server needed to proxy the request to the Origin Server to get the resource.

Let's try it again:

```
# GET curl request and selected headers
$ curl -X GET -I 172.17.0.13/css/style.css
Date: Fri, 05 Sep 2014 23:50:48 GMT
Content-Type: text/css
Last-Modified: Fri, 05 Sep 2014 22:46:39 GMT
Expires: Sat, 05 Sep 2015 23:50:12 GMT
Cache-Control: max-age=31536000
Cache-Control: public
X-Proxy-Cache: HIT
```

We can see that I requested this within ~30 seconds of the first request. It was a cache `HIT`! This file was served from the Nginx cache server.

The `Expires` header remains unchanged, as Nginx simply returned the resource from it's cache. Those headers will update the next time the Cache Server goes back to the Origin Server to get a fresh copy of the file.

Cache Control

As this stands now, the Nginx cache server will ignore a client's `Cache-Control` **request** header. This means that if our curl command said "Return me an uncached version", Nginx would ignore it.

However, we want our Cache Server to account for web clients which specify that they don't want a cached item.

For example if use our browser and hold down SHIFT while clicking the reload button, our browser will send a `Cache-Control: no-cache` request header. This asks the Cache Server to NOT serve a cached version of the resource. Our setup right now will ignore that.

In order to properly bypass the cache when requested to, we can add the `proxy_cache_bypass $http_cache_control` directive to our Cache Server in the `location` block:

```
location / {
    proxy_cache my_zone;
    proxy_cache_bypass  $http_cache_control;
    add_header X-Proxy-Cache $upstream_cache_status;

    include proxy_params;
    proxy_pass http://172.17.0.18:9000;
}
```

After saving and reloading Nginx's configuration, we can test that this works:

```
$ curl -X GET -I 172.17.0.13/css/style.css
...
X-Proxy-Cache: HIT     # A regular request which is normally a cache HIT ...

$ curl -X GET -I -H "Cache-Control: no-cache" 172.17.0.13/css/style.css
...
X-Proxy-Cache: BYPASS  # ... is now bypassed when told to
```

The `proxy_cache_bypass` directive will inform Nginx to honor the `Cache-Control` header in HTTP requests.

Proxy Caching

Nginx can also cache the results of FastCGI and uWSGI proxied requests and even the results of load balanced requests (requests sent "upstream"). This means we can cache responses from dynamic applications.

If we use Nginx to cache the results of a FastCGI process, we can think of the FastCGI process as the Origin Server and Nginx as the Cache Server. For example, on `fideloper.com` I cache the HTML results given back from PHP-FPM.

Here's an example of using `fastcgi_cache`, with some new additions:

```
fastcgi_cache_path /tmp/cache levels=1:2 keys_zone=fideloper:100m inactive=60m;
fastcgi_cache_key "$scheme$request_method$host$request_uri";

server {

    # Boilerplay omitted, such as root, server_name, etc

    set $no_cache 0;

    # Example: Don't cache admin area
    if ($request_uri ~* "/(admin/)")
    {
        set $no_cache 1;
    }

    # In Nginx 1.6.3+ you may edit this PHP
    # block in snippets/fastcgi-php.conf
    location ~ ^/(index)\.php(/|$) {
        fastcgi_cache fideloper;
```

```
20
21          # Only cache 200 responses
22          # Cache for 60 minutes
23          fastcgi_cache_valid 200 60m;
24
25          # Only GET and HEAD methods apply
26          fastcgi_cache_methods GET HEAD;
27
28          add_header X-Fastcgi-Cache $upstream_cache_status;
29
30          # Don't pull from cache based on $no_cache
31          fastcgi_cache_bypass $no_cache;
32
33          # Don't save to cache based on $no_cache
34          fastcgi_no_cache $no_cache;
35
36          # Regular PHP-FPM stuff
37          # include fastcgi_params for nginx < 1.6.1
38          include fastcgi.conf;
39          fastcgi_split_path_info ^(.+\.php)(/.+)$;
40          fastcgi_pass unix:/var/run/php5-fpm.sock;
41          fastcgi_index index.php;
42      }
43  }
```

For using caching with `FastCGI` cache, I did the following:

- Replaced all instances of `proxy_cache` with `fastcgi_cache`
- Used `fastcgi_cache_valid 200 60m` to set the expiration times on `200 OK` responses received from PHP requests sent to PHP-FPM

You can see this in action:

```
$ curl -X GET -I fideloper.com/index.php
...
Cache-Control: max-age=86400, public
X-Fastcgi-Cache: MISS

$ curl -X GET -I fideloper.com/index.php
...
X-Fastcgi-Cache: HIT

# If this URL existed, you'd see a BYPASS
$ curl -X GET -I fideloper.com/admin
...
X-Fastcgi-Cache: BYPASS
```

We can do the same for caching uWSGI responses by simply switching `proxy_cache` or `fastcgi_cache` with `uwsgi_cache` directives!

We also did a few intersting things above.

fastcgi_cache_valid

This sets how long the cache is valid for certain HTTP responses. `fastcgi_cache_valid 200 60m;` sets the cache for 200 HTTP responses to 60 minutes.

We set multiple HTTP response codes. Setting it to `fastcgi_cache_valid 200 302 10m;` will save the result of 200 and 302 HTTP responses for 10 minutes.

fastcgi_cache_methods

This allows us to set which HTTP methods are valid for caching. Using `fastcgi_cache_methods GET HEAD;` says to only cache results of GET or HEAD http requests.

fastcgi_cache_bypass and $no_cache

We set a `$no_cache` variable for URLs in the `/admin` area. We use this in conjunction with the `fastcgi_cache_bypass` to tell Nginx to not cache responses from admin-area requests.

fastcgi_no_cache

While `fastcgi_cache_bypass` tells Nginx to bypass the cache, the `fastcgi_no_cache` directive can tell Nginx to not cache the result of the response.

Here we use the `$no_cache` variable for both directives. We want Nginx to never cache the result of requests to `/admin` URIs, and to bypass checking the cache for such requests.

Example: Caching Specific URIs

The greatest gains from caching may come from caching dynamic content, rather than static content. This is because dynamic content is more expensive - it may do network operations, talk to databases, run calculations, and more.

Attempting to cache the results of a dynamic request may result in unwanted behavior. If your site allows users to login, they may need to be presented different data, for example. Caching the results of URIs may end up showing User A the dashboard of User B if User B's was cached.

Making this possible requires a more sophisticated caching technique - perhaps using an object cache (memcache, redis), or using something like Varnish which supports caching only portions of the returned HTML (via ESI - Edge Server Includes).

Let's cover one use case that may be of interest - caching one specific URI. We can tell Nginx to cache one URI that is normally proxied to dynamic application. All other URI's will be proxied as normal, rather than cached.

Let's adjust our example to do this:

```
fastcgi_cache_path /tmp/cache levels=1:2 keys_zone=fideloper:100m inactive=60m;
fastcgi_cache_key "$scheme$request_method$host$request_uri";

server {

    # Boilerplay omitted, root, server_name, try_files, etc

    location /expensive-widget {
        fastcgi_cache fideloper;
        fastcgi_cache_valid 200 60m; # Cache 200 responses, for 60 minutes
        fastcgi_cache_methods GET HEAD; # Only GET and HEAD methods apply
        add_header X-Fastcgi-Cache $upstream_cache_status;

        # Here we ignore headers that may cause issues in caching
        # if we want to cache this content no matter what
        fastcgi_ignore_headers Set-Cookie Cache-Control Expires;

        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php;
        include fastcgi_params;
        # Path to the php file to be used
        fastcgi_param SCRIPT_FILENAME /var/www/index.php;
        # Hardcode our URI
        fastcgi_param PATH_INFO /widget/v1/faq;
```

```
    }

    # We can pass other PHP requests to
    # PHP-FPM as normal
    #
    # In Nginx 1.6.3+ you may edit this PHP
    # block in snippets/fastcgi-php.conf
    location ~ ^/(index)\.php(/|$) {
        # Regular PHP-FPM stuff
        # include fastcgi_params for nginx < 1.6.1
        include fastcgi.conf;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php;
    }
}
```

Here we cached only URI's *starting* with `/expensive-widget`. The way this is defined, it may also cache `/expensive-widget/doodad` or something like `/expensive-widget/search?q=phrase-to-search`. In order to match an exact URI, use the following location block - note the = sign:

```
location = /expensive-widget {
    # cache and proxy items omitted
}
```

Varnish

Varnish is a fully-featured cache server with a good range of utility. It can act as a web cache and even as a load balancer.

It's a great tool, but has one large limitation: It can't handle SSL connections. This means if your site is behind SSL, then you need something in front of Varnish to terminate the HTTPS connection. This is often a reverse proxy (and/or load balancer) such as Nginx, HAProxy, or Pound.

Nginx can handle SSL and does HTTP caching. Nginx can also cache the results of requests proxied over FastCGI and uWSGI. On top of this, Varnish has been found to only be marginally faster at serving static files than Nginx.

I may be biased against Varnish (apologies for letting my bias show through!) but I don't usually reach for Varnish first. You may wonder when, if the choice is mine to make (a luxury we're no always given), I would use Varnish over Nginx.

Varnish is much more configurable than Nginx. You can get *very* granular with how, when and what gets cached. It also has some caching features Nginx lacks.

I like to use Varnish for these use cases:

- Nginx is not available or inappropriate to introduce into infrastructure (lack of support or in-house knowledge on it for example)
- You want or need to use ESI[86] to cache portions of your dynamically generated HTML
- Special cases, such as caching API requests made from within your application (perhaps for pinging an API that has a rate limit, or saving Github tarballs for faster dependency gathering or when GitHub goes down)
- You need to ignore cache rules from badly behaving origin servers you can't configure nor control
- You need a very fine-grain of control over how items are cached. Wordpress and other CMSes, for example, may output static files through a dynamic script. Varnish can handle caching those files, even if they aren't served with cache headers.

Origin Server

In the Origin Server chapter, we setup the server that will be serving files. We'll use this same server as a base for testing with Varnish.

In this chapter we'll have Varnish act as the HTTP cache and Nginx act as the origin server. Nginx will not be doing any caching.

[86] https://www.varnish-cache.org/trac/wiki/ESIfeatures

Install Varnish

First, we'll install Varnish on a server.

```
$ sudo apt-get install apt-transport-https
$ curl https://repo.varnish-cache.org/GPG-key.txt | apt-key add -
# For ubuntu 14.04 Precise!
$ echo "deb https://repo.varnish-cache.org/ubuntu/ precise varnish-4.0" | sudo t\
ee /etc/apt/sources.list.d/varnish-cache.list
$ sudo apt-get update
$ sudo apt-get install -y varnish

# Check version installed
$ varnishd -V
varnishd (varnish-4.0.3 revision b8c4a34)
Copyright (c) 2006 Verdens Gang AS
Copyright (c) 2006-2014 Varnish Software AS
```

We'll setup Varnish to run on port 80. If Nginx is on the same server, we need to set it to listen on another port. Edit your Nginx server configuration to listen on port 8080 instead:

File: /etc/nginx/sites-available/your-server-config

```
server {
    listen 8080;

    // Other boilerplate
}
```

Then run `sudo service nginx reload` to reload that configuration.

Assuming both Nginx and Varnish are now running, head to `http://your-server:6081` and you should see a result!

This takes advantage of Varnish's defaults. Varnish will listen for connections on port `6081` and expects an origin server at `127.0.0.1:8080`, where Nginx is listening.

Basic Configuration

We want Varnish to listen on port 80. To do so, we'll edit it's configuration at `/etc/defaults/varnish`. This file contains configuration used for running Varnish when the system boots.

Change the default `6081` to port `80`. It will look like so:

File: /etc/defaults/varnish

```
DAEMON_OPTS="-a :80 \
             -T localhost:6082 \
             -f /etc/varnish/default.vcl \
             -S /etc/varnish/secret \
             -s malloc,256m"
```

When that's saved, restart Varnish via `sudo service varnish restart`.

If our Nginx server was not listening on port `8080`, we would need to adjust Varnish to send requests to its location. We can define the backends (origin servers) that Varnish sends requests to within file `/etc/varnish/default.vcl`.

For example, if Nginx is on another server, listening on port `80`, we can change the defaults to something like the following:

```
backend default {
    .host = "192.168.33.10";
    .port = "80";
}
```

Note that the backend is named `default`. You can actually define multiple backends:

```
backend marketingsite {
    .host = "192.168.33.10";
    .port = "8080";
}

backend appsite {
    .host = "192.168.33.12";
    .port = "8080";
}
```

Then later in the same file, we can edit the `vcl_recv` parameters to decide which backend to send requests to. In this case, we'll decide based on the hostname used:

```
sub vcl_recv {
    if (req.http.host == "www.example.com") {
        set req.backend_hint = marketingsite;
    } else {
        set req.backend_hint = appsite;
    }
}
```

There's more information on backend configuration available here[87].

You can use Varnish as a load balancer. To do so, set multiple backends, add health checks[88] and use `directors` to decide how to direct traffic between the backends!

The documentation linked above explains how to do that.

I won't use any load balancing for this example, but that's a useful feature to keep in mind!

Varnish Configuration: VCL

VCL stands for Varnish Configuration Language. This is a DSL (domain-specific language) created by Varnish.

VCL lets you interact with HTTP requests and caching at different stages of the request/response life-cycle. There are different sub-routines that are called in various stages of this life-cycle.

While there are quite a few built-in sub-routines[89], the main ones you'll see in the `/etc/varnish/default.vcl` file are:

- **vcl_recv** - Called at the beginning of a received request. It's good for deciding if it should serve the request, pass off the request, and/or which backend (origin server) will fulfill a request. We saw this in the example above.
- **vcl_backend_response** - Called after a response has been receivd by a backend (origin server). This is a good place to do things like strip server-set cookies or modify the response from the origin server in other ways.
- **vcl_deliver** - Called before a cached object is delivered to the client. Here Varnish can do things like set a header to inform us if the response came from the cache or not.

[87] https://www.varnish-cache.org/docs/trunk/users-guide/vcl-backends.html
[88] https://www.varnish-cache.org/docs/trunk/users-guide/vcl-backends.html#health-checks
[89] https://www.varnish-cache.org/docs/4.0/users-guide/vcl-built-in-subs.html

Varnish Headers and Caching

Without any configuration, if you inspect some requests, we'll see some extra Varnish information added to the response headers.

You may see a `X-Varnish` header. The value of this header is the ID of the current request and the value of the request that populated the cache. If you see one number here, you can assume a cache miss.

We can use cURL to test this out:

```
$ curl -I http://192.168.22.10.xip.io/static/css/styles.css
HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Thu, 23 Oct 2014 01:27:48 GMT
Content-Type: text/css
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
ETag: "52b9b863-2df7"
X-Varnish: 131096
Age: 0
Via: 1.1 varnish-v4
Content-Length: 11767
Connection: keep-alive

$ curl -I http://192.168.22.10.xip.io/static/css/styles.css
HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Thu, 23 Oct 2014 01:27:48 GMT
Content-Type: text/css
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
ETag: "52b9b863-2df7"
X-Varnish: 131099 131097
Age: 3
Via: 1.1 varnish-v4
Content-Length: 11767
Connection: keep-alive
```

Note that on the second request, we get a cache hit. The `X-Varnish` header has both a request ID and the ID of the request that filled the cache.

Similar to how we tested Nginx, let's next see if Varnish will handle Etags (validation caching):

```
 curl -I \
    -H 'If-None-Match: "52b9b863-2df7"' \
    192.168.22.10.xip.io/static/css/styles.css
HTTP/1.1 304 Not Modified
Server: nginx/1.6.2
Date: Fri, 24 Oct 2014 01:35:04 GMT
Content-Type: text/css
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
ETag: "52b9b863-2df7"
Expires: Sat, 24 Oct 2015 01:35:04 GMT
Cache-Control: max-age=31536000, public
X-Varnish: 65548 65540
Age: 45
Via: 1.1 varnish-v4
Connection: keep-alive
```

Yep! We received a `Not Modified` response. Let's try that with a `If-Modified-Since` date:

```
$ curl -I \
    -H 'If-Modified-Since: Tue, 24 Dec 2013 16:37:55 GMT' \
    192.168.22.10.xip.io/static/css/styles.css
HTTP/1.1 304 Not Modified
Server: nginx/1.6.2
Date: Fri, 24 Oct 2014 01:35:04 GMT
Content-Type: text/css
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
ETag: "52b9b863-2df7"
Expires: Sat, 24 Oct 2015 01:35:04 GMT
Cache-Control: max-age=31536000, public
X-Varnish: 65554 65540
Age: 470
Via: 1.1 varnish-v4
Connection: keep-alive
```

That works too, great! We can use Varnish for both expiration and validation caching.

Some Debugging Utilities

Let's have Varnish tell us directly if we have a cache HIT or MISS. A HIT means Varnish found that the resource is cachable, the resource was previously cached, and Varnish returned the cached copy to you instead of asking for it from the origin server. A MISS means any of those three possibilities were not met and the resource was returned from the origin server.

To add this information, edit `/etc/varnish/default.vlc`. Modify the `vcl_deliver` section to add the `X-Cache` header:

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

The `obj.hits` variable is the number of hits a cached object has made. You can see more about what variables are available in which VCL section here[90].

After restarting Varnish, we'll see whether the resource came from the cache or not in the response headers.

```
$ curl -I http://192.168.22.10.xip.io/static/css/styles.css
HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Thu, 23 Oct 2014 01:27:48 GMT
Content-Type: text/css
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
ETag: "52b9b863-2df7"
X-Varnish: 131096
X-Cache: MISS
Age: 0
Via: 1.1 varnish-v4
Content-Length: 11767
Connection: keep-alive

$ curl -I http://192.168.22.10.xip.io/static/css/styles.css
HTTP/1.1 200 OK
Server: nginx/1.6.2
Date: Thu, 23 Oct 2014 01:27:48 GMT
Content-Type: text/css
Last-Modified: Tue, 24 Dec 2013 16:37:55 GMT
ETag: "52b9b863-2df7"
X-Varnish: 131099 131097
X-Cache: HIT
Age: 3
```

[90] https://www.varnish-software.com/static/book/VCL_functions.html

```
Via: 1.1 varnish-v4
Content-Length: 11767
Connection: keep-alive
```

Note that in our first request, the `styles.css` file was not fulfilled from the cache. In the subsequent request, it was.

X-Forwarded

Since Varnish is a proxy sitting between the client and origin server, let's make sure that the origin server knows the client's true IP address. For more information on why this is important, see chapter "Implications of Multi-Server Environments".

We can add the client's true IP address by adding the following to the `vcl_recv` block:

```
# set (if not already present)
# or append the client.ip to X-Forwarded-For header
if (req.http.X-Forwarded-For) {
    set req.http.X-Forwarded-For = req.http.X-Forwarded-For + ", " + client.ip;
} else {
    set req.http.X-Forwarded-For = client.ip;
}
```

This will set the `X-Forwarded-For` header sent to Nginx. If we have a web application, we can use that IP address to know who the client actually is, which may be needed within the application.

Increasing Cache Hit Rate

Our goal in using Varnish is (usually) to cache as much as possible. This means configuring Varnish to get as many cache hits as possible.

> You may be interested in the following blog and its article discussing if it's worth caching static files[91].

Cookies

Cookies are unique per user visiting your site. However, just like appending query strings to a URI (`styles.css?1234`), *a different cookie makes a resource unique.*

[91] https://ma.ttias.be/stop-caching-static-files/

That means Allison with one cookie is not getting the same cached file back from Varnish as Bob with another cookie, even if it's just a CSS file that should be the exact same for everyone. Resources with cookies are typically not cached or are cached uniquely per user!

We want Varnish to strip cookies from our static assets so that our origin server doesn't see it.

First, let's remove some headers commonly added by javascript snippets, such as Google Analytics and various ad trackers. Our servers typically ignore these, but our browsers will still send them in HTTP requests since they are set by these services.

```
sub vcl_recv {
    # Other items already added here omitted

    # Remove the "has_js" cookie
    set req.http.Cookie = regsuball(req.http.Cookie, "has_js=[^;]+(; )?", "");

    # Remove any Google Analytics based cookies
    set req.http.Cookie = regsuball(req.http.Cookie, "__utm.=[^;]+(; )?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "_ga=[^;]+(; )?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "utmctr=[^;]+(; )?", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "utmcmd.=[^;]+(; )?","");
    set req.http.Cookie = regsuball(req.http.Cookie, "utmccn.=[^;]+(; )?","");

    # Remove cookies from common ad trackers or services
    # DoubleClick
    set req.http.Cookie = regsuball(req.http.Cookie, "__gads=[^;]+(; )?", "");
    # Quant Capital
    set req.http.Cookie = regsuball(req.http.Cookie, "__qc.=[^;]+(; )?", "");
    # AddThis
    set req.http.Cookie = regsuball(req.http.Cookie, "__atuv.=[^;]+(; )?", "");
}
```

These will be removed on all requests to the origin server, whether for dynamic content or static.

Next, let's get rid of *all* cookies set in requests for common static files. While the above configuration removed only certain cookies, the below will remove *all* cookies set in requests for these static assets.

```
sub vcl_recv {
    # Other items already added here omitted

    # Remove all cookies for static files
    # Other candidates: bmp, bz2, flv, gz, doc, docx, rtf, swf, txt,
    #                   xml, eot, pdf, woff, less, gif
    if (req.url ~ "^[^?]*\.(css|gif|ico|jpeg|jpg|js|png)(\?.*)?$") {
      unset req.http.Cookie;
    }
}
```

So far we've used `vcl_recv` for requests coming into Varnish. Let's also unset cookies coming back from the origin server, using the `vcl_backend_response` block. This is good just in case the origin server returns cookies to be set for these static assets:

```
sub vcl_backend_response {
    # Other items already added here omitted

    # Remove all cookies for static files
    # Other candidates: bmp, bz2, flv, gz, doc, docx, rtf, swf, txt,
    #                   xml, eot, pdf, woff, less, gif
    if (beresp.url ~ "^[^?]*\.(css|gif|ico|jpeg|jpg|js|png)(\?.*)?$") {
        unset beresp.http.set-cookie;
    }
}
```

Most dynamic applications create cookies. Some sites, perhaps a Wordpress site, don't need the average user to have a cookie. CMSes for example, often don't need or want public users to log in or track any information for them. They therefore don't need to set any cookies for that user. Even so, some CMSes do set cookies as default behavior, "just in case you need them".

If that is your use case as well, the following is useful for stripping cookies from *all* requests that aren't at the `/admin` url. This assumes you want cookies set in the `/admin` area so you, the site administrator, can login!

You can change this to whatever you need - perhaps to `wp-admin` for a Wordpress site:

```
vcl_recv {
    # Other items already added here omitted

    # Unset cookie unless in /admin area
    if (!(req.url ~ "^/admin/")) {
        unset req.http.Cookie;
    }
}
```

Note that this is a bit of a sledge-hammer approach, since it disregards the rules we set previously. It unsets *any cookie* from incoming requests that's not in the `/admin` uri!

Force Caching

If you have assets that refuse to be cached (no cache headers sent, or poorly configured headers), you can tell Varnish to do it.

Let's say we have an image resizer script used at `/images.php?path=/path/to/file.jpg&width=200`. This will find the image set in the `path` GET variable, load the image, resize it, and return it. That's an expensive server operation!

The following will cache the result of that request for `5` days, regardless of the returned `cache-control` header (if even set) from the origin server. We use the `vcl_backend_resposne` block to capture/set that from the response returned from the origin server:

```
vcl_backend_response {
    # Other items here omitted

    if (bereq.url ~ "^/images\.php(\?.*)?$") {
        # Cache for 5 days
        set beresp.ttl = 5d;
    }
}
```

Varnish Tools

Here are some tools and other things you can do that might be helpful!

Purge

You may want to force a purge of a resource from cache. Varnish lets you do this via a `PURGE` http verb.

First, if you want, you can set an ACL (access control) so that purging can only be done from the localhost network:

```
acl purge {
    # ACL we'll use later to allow purges
    "localhost";
    "127.0.0.1";
    "::1";
}
```

Then we can use `vcl_revc` to match against the acl rule we named `purge`:

```
sub vcl_recv {
    # Other items omitted

    if (req.method == "PURGE") {
        # purge is the ACL defined at the begining
        if (!client.ip ~ purge) {
          # If not from allowed IP
          return (synth(405, "This IP is not allowed to send PURGE requests."));
        }
        # Purge if ACL allows:
        return (purge);
    }
}
```

Finally we can use the `vcl_purge` block to only purge when http verb "PURGE" is used:

```
sub vcl_purge {
    # Only handle http PURGE verb
    if (req.method != "PURGE") {
        set req.http.X-Purge = "Yes";
        return(restart);
    }
}
```

You can test this out with a curl request made on the Varnish server (since the ACL defines that you must make a request from localhost):

```
$ curl -X PURGE -i localhost:80/styles.css
HTTP/1.1 200 Purged
Date: Sat, 25 Apr 2015 19:25:55 GMT
Server: Varnish
X-Varnish: 9
Content-Type: text/html; charset=utf-8
Retry-After: 5
Content-Length: 236
Connection: keep-alive
```

Grace Periods

If your origin server becomes unavailable, Varnish will receive no response when attempting to refresh a stale resource from it. Normally Varnish will pass the HTTP error from the origin server through to the client.

However we can set a "grace" period where Varnish will keep serving the stale resource for a configured amount of time, which may keep your site (or portions of it) up while you fix the true issue.

This can be done in the `vcl_backend_response` block.

```
sub vcl_backend_response {
    # Other items already here omitted

    # Set the grace period to 6 hours
    set beresp.grace = 6h;
}
```

You can get a bit more granular with how grace periods behaves - see the documentation[92] on setting the grace period.

"Security"

If you prefer, we can unset some more headers to mask information about the origin server. We'll use the `vcl_deliver` to strip some headers before delivering the request to the client:

[92] https://www.varnish-cache.org/docs/trunk/users-guide/vcl-grace.html

```
sub vcl_deliver {
    # Other items here omitted

    # Unset some headers
    unset resp.http.X-Powered-By; # php version
    unset resp.http.Server;       # server type (Nginx, Apache, IIS)
    unset resp.http.X-Varnish;    # varnish brag
    unset resp.http.Via;          # more varnish brag
}
```

Extra Resources

- Many good examples of what you can do in Varnish configuration here[93].
- Achieving High Hitrate[94] docs from Varnish
- Turning off ETags in Nginx[95] to prevent Validation caching with ETags

[93] https://github.com/mattiasgeniar/varnish-4.0-configuration-templates/
[94] https://www.varnish-cache.org/docs/4.0/users-guide/increasing-your-hitrate.html
[95] http://nginx.org/en/docs/http/ngx_http_core_module.html#etag

Logs

Almost all application services and processes create logs. Logs can slowly eat away at your servers' hard-drive space, and so it's important to keep them under control.

This section will cover some ways of managing server logs.

Logrotate

Server software often logs events and errors to log files. For the most part, systems typically can take care of managing log files so they do not eventually eat up available hard drive space. Not all software is configured to do this, however.

Compounding this, many application frameworks have their own logging in place. Few manage the deletion or compression of their log files.

Log management primarily consists of:

- Rotating log files
- Backing up log files
- Aggregating logs in multiple-servers environments

In all cases where log files are not actively managed, you should at least set up log rotation and backup.

Logrotate is there to do that for you. It is available and used on most linux distributions by default.

What does Logrotate do?

Logrotate helps to manage your log files. It can periodically read, minimize, back up, create new log files, and run custom scripts. This is usually used to to help prevent any single log file from getting unwieldy in size, as well as delete old log files.

Many applications setup Logrotate for you. For instance, installing Apache in Ubuntu adds the file `/etc/logrotate.d/apache2`, which is a configuration files used by Logrotate to rotate all apache access and error logs.

Configuring Logrotate

In stock Debian/Ubuntu, any config file you put into `/etc/logrotate.d` is going to run once per day. Logrotate configuration files can specify how often logs should be rotated (at a period of 1 day or more by default). Apache's default in Ubuntu is set to run weekly, as we'll see shortly.

Logrotate's main configuration file is found in `/etc/logrotate.conf`.

File: /etc/logrotate.conf

```
# see "man logrotate" for details
# rotate log files weekly
weekly

# Perform actions as user `root` and group `syslog` by default
# This is the user/group of /var/log/syslog.
su root syslog

# keep 4 weeks worth of backlogs
rotate 4

# create new (empty) log files after rotating old ones
create

# uncomment this if you want your log files compressed
#compress

# packages drop log rotation information into this directory
# by including any file found in this directory
include /etc/logrotate.d

# no packages own wtmp, or btmp -- we'll rotate them here
/var/log/wtmp {
    missingok
    monthly
    create 0664 root utmp
    rotate 1
}

/var/log/btmp {
    missingok
    monthly
    create 0660 root utmp
    rotate 1
}

# system-specific logs may be configured here
```

These are global defaults. We can see that logrotate will rotate log files `weekly`, keeping `4` weeks of log files before deleting any via the `rotate 4` directive.

We can also see it includes configuration files found in `/etc/logrotate.d`.

We'll cover more options below by analyzing the default for Apache and then modifying it.

For Example: Apache

Let's look over Apache's default logrotate file in Debian/Ubuntu:

```
/var/log/apache2/*.log {
    weekly
    missingok
    rotate 52
    compress
    delaycompress
    notifempty
    create 640 root adm
    sharedscripts
    postrotate
            if /etc/init.d/apache2 status > /dev/null ; then \
                /etc/init.d/apache2 reload > /dev/null; \
            fi;
    endscript
    prerotate
            if [ -d /etc/logrotate.d/httpd-prerotate ]; then \
                    run-parts /etc/logrotate.d/httpd-prerotate; \
            fi; \
    endscript
}
```

This will rotate any files in the `/var/log/apache2` directory that end in `.log`. This is why, when we create a new virtual host, we typically put the logs in `/var/log/apache2`. Logrotate will automatically manage the log files!

Let's go through the options above:

weekly

This tells Logrotate to rotate these logs once per week. There are other times you can specify as well:

- daily
- weekly
- monthly

- yearly

Since Logrotate runs once per day by default, there's no option for rotating logs more than once per day. You can see the CRON task for Logrotate at `/etc/cron.daily/logrotate`. If you need to run Logrotate more than once per day, you can add a cron task on the `cron.hourly` directory which calls logrotate on a specific configuration file:

CRON task calling ficticious logrotate configuration /etc/logrotate.hourly.conf

```
1 # file '/etc/cron.hourly/logrotate'
2 /usr/bin/env logrotate /etc/logrotate.hourly.conf
```

missingok

If no `*.log` files are found, don't raise an error.

rotate 52

Keep 52 archived log file before deleting old log files (If rotating weekly, that's 52 weeks, or one years worth of logs!)

compress

Compress (gzip) rotated log files. There are some related directives you can use as well:

delaycompress

Delays compression until 2nd time around rotating. As a result, you'll have one current log file, one older log file which remains uncompressed, and then a series of compressed logs.

This is useful if a process (such as Apache) cannot be told to immediately close the log file for writing. It makes the old file available for writing until the next rotation.

If used, you'll see log files like this:

- access.log
- access.log.1
- access.log.1.gzip

You can see that `access.log.1` has been rotated out but is not yet compressed.

compresscmd

Set which command to used to compress. Defaults to gzip. An example usage: `compresscmd gunzip`.

uncompresscmd

Set the command to use to uncompress. Defaults to gunzip. An example usage: `uncompresscmd gunzip`.

notifempty

Don't rotate empty log files.

create 640 root adm

Create new log files with set permissions/owner/group, This example creates file with user `root` and group `adm`. In many systems, it will be `root` for owner and group.

The file mode will be set to 640, which is u=rw, g=r, o-rwx. Refer to the chapter on "Permission and User Management" for more information on setting file permissions.

postrotate

Specify scripts to run after rotating is done. In this case, Apache is reloaded so it writes to the newly created log files. Reloading Apache (gracefully) lets any current connection finish before reloading and setting the new log file to be written to.

The end of the script is denoted with the `endscript` directive.

sharedscripts

Run a `postrotate` script after *all* logs are rotated. If this directive is not set, it will run `postrotate` scripts after *each* matching file is rotated.

prerotate

Run scripts *before* log rotating begins. Just like with `postrotate`, the end of the script is denoted with the `endscript` directive.

Not here that the pre-rotate script called is `run-parts /etc/logrotate.d/httpd-prerotate;`. The `run-parts` command attempts to run any scripts within the given directory.

This `prerotate` directive is saying to find any executable scripts within `/etc/logrotate.d/httpd-prerotate` (if the directory exists) and run them, giving us an avenue to run any scripts prior to rotation simply by putting them into the `/etc/logrotate.d/httpd-prerotate` directory.

This directory may not exist. To use it, we can simply create the directory and add in any scripts we may need. Just make sure the script is owned and executable by user "root": `sudo chown root /etc/logrotate.d/httpd-prerotate/some-script.sh && sudo chmod u+x /etc/logrotate.d/httpd-prerotate/some-script.sh`.

For Example: Application Logs

Here's the Logrotate configuration I have for an application in production, which has a verbose application logger in place.

File: /etc/logrotate.d/some-app

```
/var/www/some-app/app/storage/logs/*.log {
    daily
    missingok
    rotate 7
    compress
    delaycompress
    notifempty
    create 660 www-data www-data
    sharedscripts
    dateext
    dateformat -web01-%Y-%m-%d-%s
    postrotate
        /usr/bin/aws s3 sync /var/www/some-app/app/storage/logs/*.gz s3://app_lo\
gs
    endscript
    prerotate
        if [ -d /etc/logrotate.d/httpd-prerotate ]; then \
            run-parts /etc/logrotate.d/httpd-prerotate; \
        fi; \
    endscript
}
```

As usual, we specify the configuration based on the location of the log files. In this case, we configure it to search for logs within the log directory of the application.

The other items to note in the above Logrotate configuration:

daily

As this application expects a large amount of traffic, the configuration rotates logs daily. The application logs are likely to grow quickly.

rotate 7

Keep only the last 7 days of logs in the server. We can keep this small because we'll move the logs off of the server as backup.

create 660 appuser www-data

Logs for this application are not being written to the `/var/log` directory. Additionally, new log files in this example are owned by user `www-data`. Assuming the application is run as user `www-data` as well, this setting ensures that the application can continue to write to the log files which logrotate manages.

We set the file permissions to 660, which lets the user and group read and write to the log files. This is best if you rely on group permissions so multiple users (perhaps a deployment user and the web application user of group `www-data`) can write to files as needed.

dateext

Logs by default get a number appended to their filename. This option appends a date instead.

Some related directives:

dateformat This is the format of the date appended to the log filename.

In this example - `dateformat -web01-%Y-%m-%d-%s`, Logrotate will also add "web01", "web02" (and so on) to the log file name so we know which webserver the log came from. This is recommended if you are logging on multiple web servers, likely behind a load balancer. Knowing what server the logs came from may be useful.

This naming scheme isn't dynamic but instead is hardcoded as "web01" and so forth - naming them correctly would be a exercise left to you (to do via automation or manually). Note that with log aggregators, this may not be a needed addition.

postrotate

Here we're simply backing up the log files to an Amazon S3 bucket. This uses AWS's command line tool[96], which is fairly easy to setup and use (install via Python's package manager Pip).

This script simply calls the S3 tool and "syncs" the log directory to the give S3 bucket. The "sync" command will keep the directories in sync, similar to the rsync utility.

This way we can allow Logrotate to delete old log files without losing any logs, as they are backed up to S3.

Going Further

That's it for Logrotate. It's a fairly simple utility overall. It's well-worth using in any application in production. If you are writing any applications, whether for utility or otherwise, it's good practive to prepare a Logrotate configuration for it.

[96] http://aws.amazon.com/cli/

Taking this to the next level, we can look into automatically moving log files to a central location. This can be done with rsyslog, a utility to centralize log locations, or with the plethora of open source and paid services used for managing and analyzing server and application log files.

Rsyslog

Debian and Ubuntu servers (among others) run the `rsyslog` service, which is primarily responsible for collecting log output and writing it to the right place, usually somewhere within the `/var/log` directory, but also to remote locations over the network.

Other distributions use `syslog-ng` rather than `rsyslog`, but the general idea is similar for all logs.

Not all applications and processes use rsyslog. Some applications write their own log files, notably Apache and Nginx. One application which does use rsyslog is Haproxy.

Configuration

Rsyslog's configuration can be found at `/etc/rsyslog.conf`. In Debian-based systems, this file is responsible for enabling modules (such as UDP/TCP & local system listeners, along with writers). It also sets some baseline global settings, such as the user and group rsyslog creates log files as.

Here's some configuration:

$ModLoad imuxsock

This loads in the imuxsock module, which creates a Unix socket for receiving log messages locally. This is always enabled.

$ModLoad imklog

This provides support for kernel logging. Again, this is always enabled.

$ModLoad imudp

Disabled by default on Debian/Ubuntu, this sets up UDP-based logging. This is over the network rather than being limited to local connections like with imuxsock.

This works in conjunction with the `$UDPServerRun 514` directive, which sets the UDP port to listen on to port 514.

$ModLoad imtcp

Disabled by default on Debian/Ubuntu, this sets up TCP-based logging. This is also over the network rather than being limited to local connections like with imuxsock.

This works in conjunction with the `$InputTCPServerRun 514` directive, which sets the TCP port to listen on to port 514.

$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat

This sets the log format to use when writing out logs. By default the Traditional format has precision down to the second. If you need more precision, you can just comment out this line, which defaults back to a more verbose log format.

$RepeatedMsgReduction on

Ignore duplicate log messages. I (and the documentation) recommend turning this off unless you are concerned with log file size. Many log aggregators duplicate this functionality as well, so it may not be necessary at this level.

"Off" is the default, so you can simply comment this line out to turn it off.

User/Group

Next we see this block of configuration:

```
$FileOwner syslog
$FileGroup adm
$FileCreateMode 0640
$DirCreateMode 0755
$Umask 0022
$PrivDropToUser syslog
$PrivDropToGroup syslog
```

This sets the file owner/group/permissions of the log files and directories created by rsyslog. Rsyslog's starts running as user `root` to get started, but then it will drop down to user `syslog` and group `adm` after launching. These permissions are suitable for writing to `/var/log`, without being a security concern by running as user `root`.

$WorkDirectory /var/spool/rsyslog

This is where working files are added, which are used in various ways such as a temporary location for files queued up to be sent out over a network.

$IncludeConfig /etc/rsyslog.d/*.conf

Rsyslog will include any configuration file found in the `/etc/rsyslog.d` directory which end in `.conf`.

Debian/Ubuntu servers will have a `/etc/rsyslog.d/50-default.config` file which we'll dive into next.

Facilities and Priorities (Log Levels)

After we talk configuration, we'll see how we can send logs into Rsyslog. Rsyslog has "categories" of logs, all of which you can use. These are called "facilities". Furthermore, each facility can be divided up by priority.

Facilities

The available facilities are:

Facility Number	Facility	Description
0	kern	kernel messages
1	user	user-level messages
2	mail	mail system
3	daemon	system daemons
4	auth	security/authorization messages
5	syslog	messages generated internally by syslogd
6	lpr	line printer subsystem
7	news	network news subsystem
8	uucp	UUCP subsystem
9		clock daemon
10	authpriv	security/authorization messages (old)
11	ftp	FTP daemon
12		NTP subsystem
13		log audit
14		log alert
15	cron	cron daemon
16	local0	local use 0 (local0)
17	local1	local use 1 (local1)
18	local2	local use 2 (local2)
19	local3	local use 3 (local3)
20	local4	local use 4 (local4)
21	local5	local use 5 (local5)
22	local6	local use 6 (local6)
23	local7	local use 7 (local7)

Which of these you use is actually up to you. For your own applications, you should use any of the `local*` facilities.

You don't need to use a different facility per application. Rsyslog can filter log messages based on a keyword and send it to the correct log file or network location.

Priorities

Facilities can be divvied by priorities. The available priorities are likely already familiar to you:

Numerical Code	Severity	Description
0	emerg	system is unusable
1	alert	action required immediately
2	crit	critical condition
3	error	error conditions
4	warning	warning conditions
5	notice	normal but significant conditions
6	info	informational message
7	debug	debug-level message

Default Configuration

Now we're ready to see the default configuration file found at `/etc/rsyslog.d/50-default.conf`. We'll see a bunch of facilities, priorities and we'll see the configuration for what to do with log messages sent to them.

Let's start at the top:

```
1 auth,authpriv.*                 /var/log/auth.log
2 *.*;auth,authpriv.none          -/var/log/syslog
3 kern.*                          -/var/log/kern.log
4 mail.*                          -/var/log/mail.log
```

Auth and authpriv facilities with any priority (denoted with the `*`) will go to the `/var/log/auth.log` file. If no priority is given, they'll go to the `/var/log/syslog`.

Note the use of a command with `auth,authpriv.none`. This assigns "auth" and "authpriv" with priority "none" (no priority set).

The `*.*;` parameter says that this will capture any non-defined facility and priority combination and send it to `/var/log/syslog`.

Kern and Mail logs of any priority go to their `/var/log/kern.log` and `mail.log` files respectively.

```
mail.err        /var/log/mail.err
```

Mail facility messages of the error priority will go to the `/var/log/mail.err` file.

```
news.crit                       /var/log/news/news.crit
news.err                        /var/log/news/news.err
news.notice                     -/var/log/news/news.notice
```

We see some news facilities getting explicitly defined as well for the critical, error and notice priorities.

```
*.emerg                                 :omusrmsg:*
```

Emergency priority message from any facility get sent to all logged-in users, via `omusrmsg`, the User Message Output Module.

Usage

Let's try testing these out what we've learned. We'll also learn some details about the configurations along the way.

Logging Messages From the Command Line

We can use the `logger` command to send a log message and see if it gets logged to the appropriate log file.

Let's log to the mail facility with a notice:

```
logger -p mail.notice 'this is my mail-related message'
```

We can see that any mail notice (except for those of priority "error") will get logged to `/var/log/-mail.log`. Let's check that out:

```
$ sudo tail -f /var/log/mail.log
Aug  5 00:33:38 vagrant-ubuntu-trusty-64 vagrant: this is my mail-related message
```

Great, we can see that the message was logged! Next let's add a "tag", which will add some text in each message. Perhaps we can use this for filtering later:

```
$ logger -p mail.notice -t SFH 'testing another message'
$ sudo tail -f /var/log/mail.log
Aug  5 00:33:38 vagrant-ubuntu-trusty-64 vagrant: this is my mail-related message
Aug  5 00:35:47 vagrant-ubuntu-trusty-64 SFH: testing another message
```

We can see both messages are in the log, and the 2nd one contains the tag "SFH".

Lastly, let's send a logger message to the local0 facility:

```
logger -p local0.debug -t SFH 'like, whatever dude'
```

Since we didn't define where local0 facilities or any facility with a debug priority should go, we know that these log messages will default to the `/var/log/syslog` file.

```
$ sudo tail -f /var/log/syslog
...other messages...
Aug  5 00:37:42 vagrant-ubuntu-trusty-64 SFH: like, whatever dude
```

Our message made it to the syslog!

Setting Up Custom Loggers

Finally we're ready to create a custom logger to handle logs sent to it from a specific application.

Let's create a rsyslog configuration file at `/etc/rsyslog.d/22-example.conf` for an application named "example".

Configuration files are read in alphabetical order - you'll often see a numbering convention in times like this so the order of the files are ready in can be set. We want our log definitions to be loaded before the default `50-default.config` file, and so we prepend it with 22.

Inside of `/etc/rsyslog.d/22-example.conf`, we can add the following:

File: /etc/rsyslog.d/22-example.conf

```
local0.*                /var/log/example.log
local0.err              /var/log/example.err.log
```

This will log any facility's "local0" messages to `/var/log/example.log` or error messages (and more critical) to `/var/log/example.err.log`.

We need to restart rsyslog after adding the configuration file:

```
sudo service rsyslog restart
```

Then test it out:

```
$ logger -p local0.debug -t SFH[1234] 'a debug message'
$ logger -p local0.err -t SFH[1234] 'a err message'
$ logger -p local0.crit -t SFH[1234] 'a crit message'
$ sudo cat /var/log/example.log
Aug  5 00:53:54 vagrant-ubuntu-trusty-64 SFH[1234]: a debug message
Aug  5 00:53:57 vagrant-ubuntu-trusty-64 SFH[1234]: a err message
Aug  5 00:54:01 vagrant-ubuntu-trusty-64 SFH[1234]: a crit message

$ sudo cat /var/log/example.err.log
Aug  5 00:53:57 vagrant-ubuntu-trusty-64 SFH[1234]: a err message
Aug  5 00:54:01 vagrant-ubuntu-trusty-64 SFH[1234]: a crit message
```

We can see that all logs went to the `example.log` file while error and above priorities went to the `example.err.log` file. You can divide these up so error messages don't go to the regular log like so:

File: /etc/rsyslog.d/22-example.conf

```
local0.*;local0.!err,!crit,!alert,!emerg    /var/log/example.log
local0.err                                  /var/log/example.err.log
```

Here we send all messages except error, critical alert and emergency to the `/var/log/example.log` file. Error priority messages and higher go to the `/var/log/example.err.log` file.

Remember to restart the rsyslog service after any change.

Sending Logs to Remote Servers

One thing rsyslog can do is send logs to a remote server. This is helpful for log aggregation - the receiving server can save the logs to a central location.

To do so, you must enable either your UDP or TCP based modules. TCP is "slower" because the protocol takes measure to ensure each data packet sent to a remote server is received, and re-sends them if not. UDP, however, is faster as it's a "fire and forget" protocol. If it's not important to get every single log in every case, using the UDP method may be preferred.

I'll use the TCP module. Let's say we have a receiving server at `192.168.33.10`. Inside of that server, we need to enable TCP reception. We can do that in `/etc/rsyslog.conf` by enabling TCP and setting it to listen on port 514:

File: /etc/rsyslog.conf on receiving server

```
# provides TCP syslog reception
$ModLoad imtcp
$InputTCPServerRun 1025
```

Because rsyslog is set to drop privileges from root on startup, we can't bind to ports under 1024 (all of which require sudo privileges). I have set the port number to 1025. You'll find an error in `/var/log/syslog` if you use a port number lower than 1024.

Save that configuration and restart rsyslog with `sudo service rsyslog restart`.

You can verify that something is listening on TCP port 1025:

```
$ netstat -a | grep 1025
tcp        0      0 *:1025          *:*       LISTEN
tcp6       0      0 [::]:1025       [::]:*    LISTEN
```

This shows that rsyslog is listening on all ipv4 and ipv6 networks on TCP port 1025.

Finally, on our server *creating* the logs, we can configure rsyslog to capture certain logs and send it to the receiving server:

File: /etc/rsyslog.d/22-example.conf

```
local0.*;local0.!err,!crit,!alert,!emerg    @@192.168.33.10:1025
local0.err                                  @@192.168.33.10:1025
```

Using `@@` denotes to send as TCP, while a single `@` will send over UDP.

Save that and restart rsyslog using `sudo service rsyslog restart`.

On the sending server, we can test this:

```
logger -p local0.info 'this is an info message'
logger -p local0.err 'this is an error message'
```

On the receiving server, if we tail the syslog, we'll see them:

```
$ sudo tail -f /var/log/syslog
Aug  5 20:49:52 vagrant-ubuntu-trusty-64 vagrant: this is an info message
Aug  5 20:50:17 vagrant-ubuntu-trusty-64 vagrant: this is an error message
```

On the receiving server, we can also configure a redirect of these logs to a specific log file rather than the syslog.

This is a way of getting some basic log aggregation started!

If you're interested in finding out more, consider investigating how you can filter logs per application and (using $syslogtag or $programname) or using file watching[97], so that applications creating their own log files can also use rsyslog.

Should I Use Rsyslog?

I suggest using a third party log aggregator if you can. These usually come with search, analytics and even alerting capabilities. There are free (open source) alternatives as well some paid ones.

Some of these use rsyslog (they'll configure it for you), while others skip it entirely. It's good to know what rsyslog can do in general, but I don't necessarily think it's the best way to manage your logs.

Sending To Rsyslog From An Application

While you can set up rsyslog to watch log files, the following libraries can also get you started sending logs to rsyslog directly:

- PHP - Monolog + syslog handler[98] will be able to send logs to syslog and rsyslog.
- Python - The standard library[99] can send to syslog
- Ruby - Use the SysLogLogger[100] or syslog-logger[101] gems
- Nodejs - The logger Winston has support for syslog[102]

[97] https://logtrust.atlassian.net/wiki/display/LD/File+monitoring+via+rsyslog
[98] https://github.com/Seldaek/monolog
[99] https://docs.python.org/2/library/logging.handlers.html#sysloghandler
[100] https://rubygems.org/gems/SysLogLogger
[101] https://rubygems.org/gems/syslog-logger
[102] https://www.npmjs.org/package/winston-rsyslog

File Management, Deployment & Configuration Management

Managing Files

There's quite a few ways of copying files using the command line. Of course we can copy files inside of our own computer, but often we need to copy files over a network to other servers. There's a few strategies for doing so, which we'll cover here in a little more detail.

Copying Files Locally

If we only need to copy files locally, we can use the `cp` command:

Copy a file:

```
cp /path/to/source/file.ext /path/to/destination/

# To rename the file while copying it
cp /path/to/source/file.ext /path/to/destination/new-filename.ext
```

To copy a directory, we must copy recursively with the `-r` flag:

```
cp -r /path/to/source/dir /path/to/destination
# Result: /path/to/destination/dir exists!
```

SCP: Secure Copy

Secure Copy is just like the `cp` command, but it uses SSH, which is a secure method of sending data.

To copy a file to a remote server:

```
# Copy a file:
scp /path/to/source/file.ext username@hostname.com:/path/to/destination/file.ext

# To copy a directory, use the recursive flag:
scp -r /path/to/source/dir username@server-host.com:/path/to/destination
```

This will attempt to connect to `hostname.com` as user `username`. It will ask you for a password if there's no SSH key setup. If the connection is authenticated successfully, the file will be copied to the remote server.

Since this works just like SSH (using SSH, in fact), we can add flags normally used with the SSH command as well. For example, you can add the `-v` and/or `-vvv` to get various levels of verbosity in output about the connection attempt and file transfer.

You can also use the `-i` (identity file) flag to specify an SSH identity file to use:

```
scp -i ~/.ssh/some_identity.pem \
        /path/to/source/file.ext \
        username@hostname:/path/to/destination/file.ext
```

Other common options for scp:

- `-p` (lowercase) - Show estimated time and connection speed while copying
- `-P` - Choose an alternate port
- `-c` (lowercase) - Choose another cypher other than the default `AES-128` for encryption
- `-C` - Compress files before copying, for faster upload speeds (already compressed files are not compressed further)
- `-l` - Limit bandwidth used in kilobits per second (8 bits to a byte!).
  - e.g. Limit to 50 KB/s: `scp -l 400 ~/file.ext user@host.com:~/file.ext`
- `-q` - Quiet output

`-l` is an important flag, as scp can eat a lot of bandwidth if not controlled

Rsync: Sync Files Across Hosts

Rsync is another secure way to transfer files. Rsync has the ability to detect file differences, giving it the opportunity to save bandwidth and time when transferring files by only sending the difference.

Just like `scp`, `rsync` uses SSH to connect to remote hosts and send/receive files from them. For the most part, the same rules and SSH-related flags apply for `rsync` as well.

Copy files to a remote server:

```
# Copy a file
rsync /path/to/source/file.ext username@hostname.com:/path/to/destination/file.e\
xt

# To copy a directory, use the recursive flag:
rsync -r /path/to/source/dir username@hostname.com:/path/to/destination/dir
```

To use a specific SSH identity file and/or SSH port, we need to do a little more work than we did with `scp`. We'll use the `-e` flag, which lets us choose/modify the remote shell program (SSH and its options) used to send files.

```
# Send files over SSH on port 8888 using a specific identity file:
rsync -e 'ssh -p 8888 -i /home/username/.ssh/some_identity.pem' \
         /source/file.ext \
        username@hostname:/destination/file.ext
```

Other common options for rsync:

- `-v` - Verbose output
- `-z` - Compress files
- `-c` - Compare files based on checksum instead of mod-time (create/modified timestamp) and size
- `-r` - Recursive
- `-S` - Handle sparse files[103] efficiently
- Symlinks:
  - `-l` - Copy symlinks as symlinks
  - `-L` - Transform symlink into referent file/dir (copy the actual file)
- `-p` - Preserve permissions
- `-h` - Output numbers in a human-readable format
- `--exclude=""` - Files to exclude
  - e.g. Exclude the .git directory: `--exclude=".git"`

There are many other options[104] as well - you can do a LOT with rsync!

Doing a Dry-Run:

I often do a dry-run of rsync to preview what files will be copied over. This is useful for making sure your flags are correct and you won't overwrite files you don't wish to:

For this, we can use the `-n` or `--dry-run` flag:

```
# Copy the current directory
$ rsync -vzcrSLhp --dry-run \
        ./ \
        username@hostname.com:/var/www/some-site.com

building file list ... done
... list of directories/files and some meta data here ...
```

[103] http://gergap.wordpress.com/2013/08/10/rsync-and-sparse-files/
[104] http://linux.die.net/man/1/rsync

Resuming a Stalled Transfer:

Once in a while a large file transfer might stall or fail (while either using `scp` or `rsync`). We can use rsync to finish a file transfer!

For this, we can use the `--partial` flag, which tells rsync to not delete partially transferred files but keep them and attempt to complete the file's transfer:

```
rsync --partial --progress \
      largefile.ext \
      username@hostname:/path/to/largefile.ext
```

The Archive Option:

There's also a `-a` or `--archive` option, which is a handy shortcut for the options `-rlptgoD`:

- `-r` - Copy recursively
- `-l` - Copy symlinks as symlinks (don't copy the actual file)
- `-p` - Preserve permissions
- `-t` - Preserve modification times
- `-g` - Preserve group
- `-o` - Preserve owner (User needs to have permission to change owner)
- `-D` - Preserve special/device files[105]. Same as `--devices --specials`. (User needs permissions to do so)

For example (note the use of `--stats` as well):

```
# Copy using the archive option and print some stats
rsync -a --stats /source/dir/path username@hostname:/destination/dir/path
```

Smartly Merge between Directories

Rsync can be used to smartly merge two directories:

```
rsync -abviuzP src/ dest/
```

- `-i` - turns on the itemized format, which shows more information than the default format
- `-b` - makes rsync backup files that exist in both folders, appending ~ to the old file. You can control this suffix with –suffix .suf

[105] http://en.wikipedia.org/wiki/Device_file

- `-u` - makes rsync transfer skip files which are newer in dest than in src
- `-z` - turns on compression, which is useful when transferring easily-compressible files over slow links
- `-P` - turns on –partial and –progress
- `--partial` - makes rsync keep partially transferred files if the transfer is interrupted
- `--progress` - shows a progress bar for each transfer, useful if you transfer big files

Deployment

SCP and Rsync make for good but basic tools for deploying files to servers.

Currently the serversforhackers.com site is built with static files. I use a static site generator (Sculpin) to create the files, and simply use rsync to copy them to the production server. The script to do so looks something like this:

```
# Generate production files
php sculpin.phar generate --env=prod

# Upload files via rsync
rsync -vzrS output_prod/ \
      username@serversforhackers.com:/var/www/serversforhackers.com/public
```

We can do the same with SCP as well, however Rsync provides the benefit of only sending files that have changed.

This is good for basic sites, however sites that need further done to it on deployment (perhaps updating packages or reload web servers) deserve a more automated method of deployment.

Auto-deploy with GitHub

In the past, I've needed to automate deploying new code without being able to install Git on the production server. The project was hosted on GitHub, so I had GitHub's WebHooks available to me.

Node makes creating HTTP listeners very easy. Because of that, and the strength of the Node community, I first checked out what Node projects were available for receiving Github WebHooks.

I chose gith[106], which is a simple package for responding to WebHooks. Its last commit was in 2013, so you may want to find an updated library, but it will work for our example here.

How it Works

When a commit is pushed to GitHub, a POST request will be sent to a URL of our choosing. That URL is set in the "settings" page of any GitHub repository. This POST request will include a "payload" variable with information about the repository and the latest commit(s).

Our code will then take action on this - in this case, if the push was to the master branch, it will run a shell script to download the latest zip file of the repo, unzip it and move it to where it needs to be on the server. This avoids using git directly on the server, although you can do so if it fits your needs.

Node will create the web server to listen for the WebHook. It can then execute the shell script which does the heavy lifting.

Node Listener

Assuming Node and NPM are installed, we can do the following:

```
1 cd /path/to/node/app
2 npm install gith
```

`Gith` is now installed at `/path/to/node/app`, so let's write our node script using it. The node script:

Create the file `/path/to/node/app/hook.js` and edit it:

[106] https://github.com/danheberden/gith

File: /path/to/node/app/hook.js

```
// Listen on port 9001
var gith = require('gith').create( 9001 );
// Import execFile, to run our bash script
var execFile = require('child_process').execFile;

gith({
    repo: 'fideloper/example'
}).on( 'all', function( payload ) {
    if( payload.branch === 'master' )
    {
        // Exec a shell script
        execFile('/path/to/hook.sh', function(error, stdout, stderr) {
                    // Log success or error in some manner
                    console.log( 'exec complete' );
                }
        );
    }
});
```

This will run the file as the user that starts/owns the Node process. You'll want the Node process to be a user with permission to run these operations, likely your deploy user.

Buffer Size

If your shell script outputs a lot of data to stdout, then you may max out Node's "maxBuffer" setting. If this is reached, then the child process is killed! In the example above, this means that the `hook.sh` script will stop mid-process.

In order to increase the default buffer size limit, you can pass in some options to the execFile[107] function:

[107] http://nodejs.org/api/child_process.html#child_process_child_process_execfile_file_args_options_callback

```
// Increase maxBuffer from 200*1024 to 1024*1024
var execOptions = {
     maxBuffer: 1024 * 1024 // 1mb
}

// Pass execOptions
execFile('/path/to/hook.sh', execOptions,
          function(error, stdout, stderr) { ... }
```

Shell Script

We use a shell script to get the files from the `master` branch of the repository and replace the latest files with them.

Install `unzip` if you don't already have it. On Ubuntu, you can run:

```
sudo apt-get install unzip
```

Now, create the `hook.sh` shell script:

File: /path/to/node/app/hook.sh

```
#!/usr/bin/env bash

# First, get the zip file
cd /path/to/put/zip/file && wget \
   -O projectmaster.zip \
   -q https://github.com/fideloper/example/archive/master.zip

# Second, unzip it, if the zip file exists
if [ -f /path/to/put/zip/file/projectmaster.zip ]; then
    # Unzip the zip file
    unzip -q /path/to/put/zip/file/projectmaster.zip

    # Delete zip file
    rm /path/to/put/zip/file/projectmaster.zip

    # Rename project directory to desired name
    mv Project-master somesite.com

    # Delete current directory
    rm -rf /var/www/somesite.com
```

```
21
22      # Replace with new files
23      mv somesite.com /var/www/
24
25      # Perhaps call any other scripts you need to rebuild assets here
26      # or set owner/permissions
27      # or confirm that the old site was replaced correctly
28  fi
```

Putting it together

So, we have a GitHub Webhook sending POST data to `http://somesite.com:9001`, as set in GitHub project settings and in our Node script. When that hook is received, we check if it's the `master` branch. If so, we run the shell script `hook.sh`.

Lastly, We need to keep the Node script running. If it stops running without us knowing about it, then GitHub WebHook's will do nothing and we'll be running out-of-date code. This is where forever[108] comes in - It will watch a Node process and turn it back on if the Node app errors out or otherwise stops running.

```
1  # To install globally, run as a priviledged user (use sudo)
2  sudo npm install -g forever
3
4  # Start our Node app … FOREVER!
5  forever start /path/to/node/app/hook.js
```

I suggest using Supervisord or PM2 in production, as Forever isn't built to reload processes through a system restart. The Monitoring Processes chapter covers this.

Firewall

If you're using a firewall such as `iptables` , you will likely need to open your chosen port to receive web traffic. Here's how you can do it with iptables:

```
1  # (I)nserts this rule after the 4th iptables firewall rule
2  sudo iptables -I INPUT 4 -p tcp --dport 9001 -j ACCEPT
```

[108] https://github.com/nodejitsu/forever

Note that I use `-I` to insert a new rule in with existing ones. This will add it after the 4th rule. The order is important in iptables rules, since the firewall will stop and apply at the first rule that matches the incoming request.

Adding a new rule instead of inserting one can be added in this manner:

```
1 iptables -A INPUT -p tcp --dport 9001 -j ACCEPT
```

Configuration Management with Ansible

Ansible is a configuration management and provisioning tool, similar to Chef, Puppet or Salt.

I've found it to be one of the simplest and the easiest to get started with. A lot of this is because it's "just SSH"; It uses SSH to connect to servers and run the configured Tasks.

One nice thing about Ansible is that it's very easy to convert bash scripts (still a popular way to accomplish configuration management) into Ansible Tasks. Since it's primarily SSH based, it's not hard to see why this might be the case - Ansible ends up running the same commands.

We could just script our own provisioners, but Ansible is much cleaner because it automates the process of getting *context* before running Tasks. With this context, Ansible is able to handle most edge cases - the kind we usually take care of with longer and increasingly complex scripts.

Ansible Tasks are idempotent, meaning we can run the same set of tasks over and over again without worrying about negative consequences. Without a lot of extra coding, bash scripts are usually **not** safely run again and again.

To accomplish idempotence, Ansible uses "Facts", which is system and environment information it gathers ("context") before running Tasks. These facts are used to check system state and see if it needs to change anything in order to get the desired outcome.

Here I'll show how easy it is to get started with Ansible. We'll start at a basic level and then add in more features as we improve upon our configurations.

Install

Of course we need to start by installing Ansible. Tasks can be run off of any machine Ansible is installed on.

This means there's usually a "central" server running Ansible commands, although there's nothing particularly special about what server Ansible is installed on. Ansible is "agentless" - there's no central agent(s) running on the servers that are being provisioned. We can even run Ansible from any server; I often run Tasks from my laptop.

Here's how to install Ansible on Ubuntu 14.04. We'll use the easy-to-remember `ppa:ansible/ansible` repository as per the official docs[109].

[109] http://docs.ansible.com/intro_installation.html#latest-releases-via-apt-ubuntu

Installing Ansible from official repository

```
sudo apt-add-repository -y ppa:ansible/ansible
sudo apt-get update
sudo apt-get install -y ansible
```

Managing Servers

Ansible has a default inventory file used to define which servers it will be managing. After installation, there's an example one you can reference at `/etc/ansible/hosts`.

I usually *move* (rather than delete) the default one so I can reference it later:

```
sudo mv /etc/ansible/hosts /etc/ansible/hosts.orig
```

Then I create my own inventory file from scratch. After moving the example inventory file, create a new `/etc/ansible/hosts` file, and define some servers to manage. Here we'll define two servers under the "web" label:

File: /etc/ansible/hosts

```
[web]
192.168.22.10
192.168.22.11
```

That's good enough for now. If needed, we can define ranges of hosts, multiple groups, reusable variables, and use other fancy setups[110], including creating a dynamic inventory[111].

For testing this chapter, I created a virtual machine, installed Ansible, and then ran Ansible Tasks **directly on that server**. To do this, my `hosts` inventory file simply looked like this:

```
[local]
127.0.0.1
```

This makes testing pretty easy - I don't need to setup multiple servers or virtual machines. A consequence of this is that I need to tell Ansible to run Tasks as user "vagrant" and use password-based (rather than key-based) authentication.

Note what we're doing here - I'm installing Ansible on the same server I want to provision. This is not a typical setup, but is useful for testing Ansible yourself within a Virtual Machine.

[110] http://docs.ansible.com/intro_inventory.html
[111] http://docs.ansible.com/intro_dynamic_inventory.html

Basic: Running Commands

Once we have an inventory configured, we can start running Tasks against the defined servers.

Ansible will assume you have SSH access available to your servers, usually based on SSH-Key. Because Ansible uses SSH, the server it's on needs to be able to SSH into the inventory servers. It will attempt to connect as the current user it is being run as. If I'm running Ansible as user `vagrant`, it will attempt to connect as user `vagrant` on the other servers.

If Ansible can directly SSH into the managed servers, we can run commands without too much fuss:

Using the ping module

```
$ ansible all -m ping
127.0.0.1 | success >> {
    "changed": false,
    "ping": "pong"
}
```

We can see the output we get from Ansible is some JSON which tells us if the Task made any changes and the result.

If we need to define the user and perhaps some other settings in order to connect to our server, we can. When testing locally on Vagrant, I use the following:

using the ping module while using sudo, asking for user password and defining the user

```
ansible all -m ping -s -k -u vagrant
```

Let's cover these commands:

- `all` - Use all defined servers from the inventory file
- `-m ping` - Use the "ping" module, which simply runs the `ping` command and returns the results
- `-s` - Use "sudo" to run the commands
- `-k` - Ask for a password rather than use key-based authentication
- `-u vagrant` - Log into servers using user `vagrant`

Modules

Ansible uses "modules" to accomplish most of its Tasks. Modules can do things like install software, copy files, use templates and much more[112].

Modules are *the* way to use Ansible, as they can use available context ("Facts") in order to determine what actions, if any need to be done to accomplish a Task.

If we didn't have modules, we'd be left running arbitrary shell commands like this:

[112] http://docs.ansible.com/modules_by_category.html

Installing Nginx with an arbitrary shell command

```
ansible all -s -m shell -a 'apt-get install nginx'
```

Here, the `sudo apt-get install nginx` command will be run using the "shell" module. The `-a` flag is used to pass any arguments to the module. I use `-s` to run this command using `sudo`.

However this isn't particularly powerful. While it's handy to be able to run these commands on all of our servers at once, we still only accomplish what any bash script might do.

If we used a more appropriate module instead, we can run commands with an assurance of the result. Ansible modules ensure indempotence - we can run the same Tasks over and over without affecting the final result.

For installing software on Debian/Ubuntu servers, the "apt" module will run the same command, but ensure idempotence.

Installing Nginx with the pkg module

```
ansible all -s -m apt -a 'pkg=nginx state=installed update_cache=true'
127.0.0.1 | success >> {
    "changed": false
}
```

This will use the apt module[113] to update the repository cache and install Nginx (if not installed).

The result of running the Task was `"changed": false`. This shows that there were no changes; I had already installed Nginx. I can run this command over and over without worrying about it affecting the desired result.

Going over the command:

- `all` - Run on all defined hosts from the inventory file
- `-s` - Run using sudo
- `-m apt` - Use the apt module[114]
- `-a 'pkg=nginx state=installed update_cache=true'` - Provide the arguments for the apt module, including the package name, our desired end state and whether to update the package repository cache or not

We can run all of our needed Tasks (via modules) in this ad-hoc way, but let's make this more managable. We'll move this Task into a Playbook, which can run and coordinate multiple Tasks.

[113] http://docs.ansible.com/apt_module.html
[114] http://docs.ansible.com/apt_module.html

Basic Playbook

Playbooks[115] can run multiple Tasks and provide some more advanced functionality that we would miss out on if using ad-hoc commands. Let's move the above Task into a playbook.

Playbooks and Roles in Ansible all use Yaml.

Create the file `nginx.yml`:

Playbook file nginx.yml

```
1  ---
2  - hosts: local
3    tasks:
4      - name: Install Nginx
5        apt: pkg=nginx state=installed update_cache=true
```

This Task does exactly the same as our ad-hoc command, however I chose to specify my "local" group of servers rather than "all". We can run it with the `ansible-playbook` command:

Output from running the Nginx Playbook

```
1  $ ansible-playbook -s nginx.yml
2
3  PLAY [local] ******************************************************************
4
5  GATHERING FACTS ***************************************************************
6  ok: [127.0.0.1]
7
8  TASK: [Install Nginx] *********************************************************
9  ok: [127.0.0.1]
10
11 PLAY RECAP ********************************************************************
12 127.0.0.1                  : ok=2    changed=0    unreachable=0    failed=0
```

Use `-s` to tell Ansible to use `sudo` again, and then pass the Playbook file.

Alternatively, we could tell Ansible to use "sudo" from within the Playbook:

[115] http://docs.ansible.com/playbooks_intro.html

Playbook file nginx.yml

```
---
- hosts: local
  sudo: yes
  tasks:
   - name: Install Nginx
     apt: pkg=nginx state=installed update_cache=true
```

Then we could run it with the following, simpler, command:

```
$ ansible-playbook nginx.yml
```

In any case, we get some useful feedback while this runs, including the Tasks Ansible runs and their result. Here we see all ran OK, but nothing was changed. I happen to have Nginx installed already.

I used the command `$ ansible-playbook -s -k -u vagrant nginx.yml` to run this playbook locally on my Vagrant installation while testing.

Handlers

A Handler is exactly the same as a Task (it can do anything a Task can), but it will run when called by another Task. You can think of it as part of an Event system; A Handler will take an action when called by an event it listens for.

This is useful for "secondary" actions that might be required after running a Task, such as starting a new service after installation or reloading a service after a configuration change.

Adding a Handler

```
---
- hosts: local
  sudo: yes
  tasks:
   - name: Install Nginx
     apt: pkg=nginx state=installed update_cache=true
     notify:
      - Start Nginx

  handlers:
   - name: Start Nginx
     service: name=nginx state=started
```

We can add a `notify` directive to the installation Task. This notifies any Handler named "Start Nginx" after the Task is run.

Then we can create the Handler called "Start Nginx". This Handler is the Task called when "Start Nginx" is notified.

This particular Handler uses the Service module[116], which can start, stop, restart, reload (and so on) system services. Here we simply tell Ansible that we want Nginx to be started.

> Note that Ansible has us define the ***state*** you wish the service to be in, rather than defining the ***change*** you want. Ansible will decide if a change is needed, we just tell it the desired result.

Let's run this Playbook again:

Output of running the Nginx Playbook with the Handler

```
# -s flag is actually redundant with "sudo: yes" in the yaml
$ ansible-playbook -s nginx.yml

PLAY [local] ******************************************************************

GATHERING FACTS ***************************************************************
ok: [127.0.0.1]

TASK: [Install Nginx] *********************************************************
ok: [127.0.0.1]

NOTIFIED: [nginx | Start Nginx] ***********************************************
ok: [127.0.0.1]

PLAY RECAP ********************************************************************
127.0.0.1                  : ok=2    changed=0    unreachable=0    failed=0
```

We get the similar output, but this time the Handler was run.

Notifiers are only run if the Task is run. If I already had Nginx installed, the Install Nginx Task would not be run and the notifier would not be called.

We can use Playbooks to run multiple Tasks, add in variables, define other settings and even include other playbooks.

[116] http://docs.ansible.com/service_module.html

More Tasks

Next we can add a few more Tasks to this Playbook and explore some other functionality.

```
---
- hosts: local
  sudo: yes
  vars:
   - docroot: /var/www/serversforhackers.com/public
  tasks:
   - name: Add Nginx Repository
     apt_repository: repo='ppa:nginx/stable' state=present
     register: ppastable

   - name: Install Nginx
     apt: pkg=nginx state=installed update_cache=true
     when: ppastable|success
     register: nginxinstalled
     notify:
      - Start Nginx

   - name: Create Web Root
     when: nginxinstalled|success
     file: dest={{ docroot }} mode=775 state=directory owner=www-data group=www-\
data
     notify:
      - Reload Nginx

  handlers:
   - name: Start Nginx
     service: name=nginx state=started

    - name: Reload Nginx
      service: name=nginx state=reloaded
```

There are now three Tasks:

- `Add Nginx Repository` - Add the Nginx stable PPA to get the latest stable version of Nginx, using the apt_repository module[117].
- `Install Nginx` - Install Nginx using the Apt module.

[117] http://docs.ansible.com/apt_repository_module.html

- `Create Web Root` - Finally, create a web root directory.

Also new here are the `register` and `when` directives. These tell Ansible to run a Task **when** something else happens.

The "Add Nginx Repository" Task registers "ppastable". Then we use that to inform the Install Nginx Task to only run when the registered "ppastable" Task is successful. This allows us to conditionally stop Ansible from running a Task.

We also use a variable. The `docroot` variable is defined in the `var` section. It's then used as the destination argument of the file module[118] which creates the defined directory.

This playbook can be run with the usual command:

```
ansible-playbook -s nginx.yml

# Or, as I ran on my Vagrant machine:
ansible-playbook -s -k -u vagrant nginx.yml
```

Next we'll take Ansible further and by organizing the Playbook into a Role while also showing some more functionality.

Roles

Roles are good for organizing multiple, related Tasks and encapsulating data needed to accomplish those Tasks. For example, installing Nginx may involve adding a package repository, installing the package and setting up configuration. We've seen installation in action in a Playbook, but once we start configuring our installations, the Playbooks tend to get a little more busy.

The configuration portion often requires extra data such as variables, files, dynamic templates and more. These tools can be used with Playbooks, but we can do better immediately by **organizing related Tasks and data into one coherent structure: a Role**.

Roles have a directory structure like this:

[118] http://docs.ansible.com/file_module.html

Role directory structure

```
rolename
    /files
    /handlers
    /meta
    /templates
    /tasks
    /vars
```

Within each directory, Ansible will search for and read any Yaml file called `main.yml` automatically.

We'll break apart our `nginx.yml` file and put each component within the corresponding directory to create a cleaner and more complete provisioning toolset.

Files

First, within the `files` directory, we can add files that we'll want copied into our servers. For Nginx, I often copy H5BP's Nginx component configurations[119]. I simply download the latest from Github, make any tweaks I want, and put them into the `files` directory.

H5BP directory included with the Role's files

```
nginx
    /files
        /h5bp
            /-other configs from H5BP-
```

As we'll see, these configurations will be added via the copy module[120].

Handlers

Inside of the `handlers` directory, we can put all of our Handlers that were once within the `nginx.yml` Playbook.

[119] https://github.com/h5bp/server-configs-nginx/tree/master/h5bp
[120] http://docs.ansible.com/copy_module.html

File: handlers/main.yml

```
---
- name: Start Nginx
  service: name=nginx state=started

- name: Reload Nginx
  service: name=nginx state=reloaded
```

Once these are in place, we can reference them from other files freely.

Meta

The `main.yml` file within the `meta` directory contains Role meta data, including dependencies.

If this Role depended on another Role, we could define that here. For example, I have the Nginx Role depend on the SSL Role, which installs SSL certificates.

File: meta/main.yml

```
---
dependencies:
  - { role: ssl }
```

If I called the "nginx" Role, it would attempt to first run the "ssl" Role.

Otherwise we can omit this file, or define the Role as having no dependencies:

File: meta/main.yml

```
---
dependencies: []
```

Template

Template files can contain template variables, based on Python's Jinja2 template engine[121]. Files in here should end in the `.j2` extension, but can otherwise have any name. Similar to `files`, we won't find a `main.yml` file within the `templates` directory.

Here's an example Nginx virtual host configuration. Note that it uses some variables which we'll define later in the `vars/main.yml` file.

[121] http://jinja.pocoo.org/docs/dev/

File: templates/serversforhackers.com.j2

```
server {
    # Enforce the use of HTTPS
    listen 80 default_server;
    server_name *.{{ domain }};
    return 301 https://{{ domain }}$request_uri;
}

server {
    listen 443 default_server ssl;

    root /var/www/{{ domain }}/public;
    index index.html index.htm index.php;

    access_log /var/log/nginx/{{ domain }}.log;
    error_log  /var/log/nginx/{{ domain }}-error.log error;

    server_name {{ domain }};

    charset utf-8;

    include h5bp/basic.conf;

    ssl_certificate           {{ ssl_crt }};
    ssl_certificate_key       {{ ssl_key }};
    include h5bp/directive-only/ssl.conf;

    location /book {
        return 301 http://book.{{ domain }};
    }

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

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

    location ~ \.php$ {
        fastcgi_split_path_info ^(.+\.php)(/.+)$;

        fastcgi_pass unix:/var/run/php5-fpm.sock;
```

```
        fastcgi_index index.php;

        include fastcgi_params; # fastcgi.conf for version 1.6.1+
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO       $fastcgi_path_info;
        fastcgi_param ENV production;
    }

    # Nginx status
    # Nginx Plus only
    #location /status {
    #    status;
    #    status_format json;
    #    allow 127.0.0.1;
    #    deny all;
    #}

    location ~ ^/(fpmstatus|fpmping)$ {
        access_log off;
        allow 127.0.0.1;
        deny all;
        include fastcgi_params; # fastcgi.conf for version 1.6.1+
        fastcgi_pass unix:/var/run/php5-fpm.sock;
    }
}
```

This is a fairly standard Nginx configuration for a PHP application. There are three variables used here:

- domain
- ssl_crt
- ssl_key

These three will be defined in the variables section.

Variables

Before we look at the Tasks, let's look at variables. The `vars` directory contains a `main.yml` file which simply lists variables we'll use. This provides a convenient place for us to change configuration-wide settings.

Here's what the `vars/main.yml` file might look like:

File: vars/main.yml

```
---
domain: serversforhackers.com
ssl_key: /etc/ssl/sfh/sfh.key
ssl_crt: /etc/ssl/sfh/sfh.crt
```

These are three variables which we can use elsewhere in this Role. We saw them used in the template above, but we'll see them in our defined Tasks as well.

Tasks

Let's finally see this all put together into a series of Tasks.

Inside of `tasks/main.yml`:

Final Tasks file using all Role functionality

```
---
- name: Add Nginx Repository
  apt_repository: repo='ppa:nginx/stable' state=present
  register: ppastable

- name: Install Nginx
  apt: pkg=nginx state=installed update_cache=true
  when: ppastable|success
  register: nginxinstalled
  notify:
    - Start Nginx

- name: Add H5BP Config
  when: nginxinstalled|success
  copy: src=h5bp dest=/etc/nginx owner=root group=root

- name: Disable Default Site
  when: nginxinstalled|success
  file: dest=/etc/nginx/sites-enabled/default state=absent

- name: Add SFH Site Config
  when: nginxinstalled|success
  register: sfhconfig
  template: src=serversforhackers.com.j2 dest=/etc/nginx/sites-available/{{ doma\
in }}.conf owner=root group=root
```

```
26
27 - name: Enable SFH Site Config
28   when: sfhconfig|success
29   file: src=/etc/nginx/sites-available/{{ domain }}.conf dest=/etc/nginx/sites-e\
30 nabled/{{ domain }}.conf state=link
31
32
33 - name: Create Web root
34   when: nginxinstalled|success
35   file: dest=/var/www/{{ domain }}/public mode=775 state=directory owner=www-dat\
36 a group=www-data
37   notify:
38     - Reload Nginx
39
40 - name: Web Root Permissions
41   when: nginxinstalled|success
42   file: dest=/var/www/{{ domain }} mode=775 state=directory owner=www-data group\
43 =www-data recurse=yes
44   notify:
45     - Reload Nginx
```

This is a longer series of Tasks, which makes for a more complete installation of Nginx. The Tasks, in order of appearance, accomplish the following:

- Add the nginx/stable repository
- Install & start Nginx, register successful installation to trigger remaining Tasks
- Add H5BP configuration
- Disable the default virtual host by removing the symlink to the `default` file from the `sites-enabled` directory
- Copy the `serversforhackers.com.conf.j2` virtual host template into the Nginx configuration
- Enable the virtual host configuration by symlinking it to the `sites-enabled` directory
- Create the web root
- Change permission for the project root directory, which is one level above the web root created previously

There's some new modules (and new uses of some we've covered), including **copy**, **template**, & **file**. By setting the arguments for each module, we can do some interesting things such as ensuring files are "absent" (delete them if they exist) via `state=absent`, or create a file as a symlink via `state=link`. You should check the docs for each module to see what interesting and useful things you can accomplish with them.

Running the Role

Before running the Role, we need to tell Ansible where our Roles are located. In my Vagrant server, they are located within `/vagrant/ansible/roles`. We can add this file path to the `/etc/ansible/ansible.cfg` file:

```
roles_path    = /vagrant/ansible/roles
```

Assuming our `nginx` Role is located at `/vagrant/ansible/roles/nginx`, we'll be all set to run this Role!

> Remove the `ssl` dependency from `meta/main.yml` before running this Role if you are following along.

Let's create a "master" yaml file which defines the Roles to use and what hosts to run them on:

Playbook file: server.yml

```
---
- hosts: all
  sudo: yes
  roles:
    - nginx
```

> In my Vagrant example, I use the host "local" rather than "all".

Then we can run the Role(s):

```
# -s option is redundant with "sudo: yes" in the yaml
ansible-playbook -s server.yml

# Or as I do with my Vagrant VM:
ansible-playbook -s -k -u vagrant server.yml
```

Here's the output from the run of the Nginx Role:

Output from Playbook using Nginx Role

```
PLAY [all] *********************************************************************

GATHERING FACTS ****************************************************************
ok: [127.0.0.1]

TASK: [nginx | Add Nginx Repository] *******************************************
changed: [127.0.0.1]

TASK: [nginx | Install Nginx] **************************************************
changed: [127.0.0.1]

TASK: [nginx | Add H5BP Config] ************************************************
changed: [127.0.0.1]

TASK: [nginx | Disable Default Site] *******************************************
changed: [127.0.0.1]

TASK: [nginx | Add SFH Site Config] ********************************************
changed: [127.0.0.1]

TASK: [nginx | Enable SFH Site Config] *****************************************
changed: [127.0.0.1]

TASK: [nginx | Create Web root] ************************************************
changed: [127.0.0.1]

TASK: [nginx | Web Root Permissions] *******************************************
ok: [127.0.0.1]

NOTIFIED: [nginx | Start Nginx] ************************************************
ok: [127.0.0.1]

NOTIFIED: [nginx | Reload Nginx] ***********************************************
changed: [127.0.0.1]

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=8   changed=7   unreachable=0    failed=0
```

Awesome, we put all the various components together into a coherent Role and now have Nginx installed and configured!

Facts

Before running any Tasks, Ansible will gather information about the system it's provisioning. These are called Facts, and include a wide array of system information such as the number of CPU cores, available ipv4 and ipv6 networks, mounted disks, Linux distribution and more.

Facts are often useful in Tasks or Template configurations. For example Nginx is commonly set to use as any worker processors as there are CPU cores. Knowing this, you may choose to set your template of the `nginx.conf` file like so:

File: templates/nginx.conf.j2 - Template for /etc/nginx/nginx.conf file

```
1 user www-data www-data;
2 worker_processes {{ ansible_processor_cores }};
3 pid /var/run/nginx.pid;
4
5 # And other configurations...
```

Or if you have a server with multiple CPU's, you can use:

File: templates/nginx.conf.j2 with multiple CPU's and cores

```
1 user www-data www-data;
2 worker_processes {{ ansible_processor_cores * ansible_processor_count }};
3 pid /var/run/nginx.pid;
4
5 # And other configurations...
```

Ansible facts all start with `anisble_` and are globally available for use any place variables can be used: Variable files, Tasks, and Templates.

Example: NodeJS

For Ubuntu, we can get the latest stable NodeJS and NPM from NodeSource, which has teamed up with Chris Lea. Chris ran the Ubuntu repository `ppa:chris-lea/node.js`, but now provides NodeJS via NodeSource packages. To that end, they have provided a shells script which installs the latest stable NodeJS and NPM on Debian/Ubuntu systems.

This shell script is found at https://deb.nodesource.com/setup[122]. We can take a look at this and convert it to the following tasks from a NodeJS Role:

[122] https://deb.nodesource.com/setup

NodeJS and NPM Role for latest stable versions, as per NodeSource and Chris Lea

```
---
- name: Ensure Ubuntu Distro is Supported
  get_url:
    url='https://deb.nodesource.com/node/dists/{{ ansible_distribution_release }}\
}/Release'
    dest=/dev/null
  register: distrosupported

- name: Remove Old Chris Lea PPA
  apt_repository:
    repo='ppa:chris-lea/node.js'
    state=absent
  when: distrosupported|success

- name: Remove Old Chris Lea Sources
  file:
    path='/etc/apt/sources.list.d/chris-lea-node_js-{{ ansible_distribution_rele\
ase }}.list'
    state=absent
  when: distrosupported|success

- name: Add Nodesource Keys
  apt_key:
    url=https://deb.nodesource.com/gpgkey/nodesource.gpg.key
    state=present

- name: Add Nodesource Apt Sources List Deb
  apt_repository:
    repo='deb https://deb.nodesource.com/node {{ ansible_distribution_release }}\
 main'
    state=present
  when: distrosupported|success

- name: Add Nodesource Apt Sources List Deb Src
  apt_repository:
    repo='deb-src https://deb.nodesource.com/node {{ ansible_distribution_releas\
e }} main'
    state=present
  when: distrosupported|success

- name: Install NodeJS
```

```
42   apt: pkg=nodejs state=latest update_cache=true
43   when: distrosupported|success
```

There's a few tricks happening there. These mirror the bash script provided by Node Source.

First we create the `Ensure Ubuntu Distro is Supported` task, which uses the `ansible_distribution_release` Fact. This gives us the Ubuntu release, such as Precise or Trusty. If the resulting URL exists, then we know our Ubuntu distribution is supported and can continue. We register `distrosupported` so we can test if this step was successfully on subsequent tasks.

Then we run a series of tasks to remove NodeJS repositories in case the system already has `ppa:chris-lea/node.js` added. These only run when if the distribution is supported via `when: distrosupported|success`. Note that most of these continue to use the `ansible_distribution_release` Fact.

Finally we get the debian source packages and install NodeJS after updating the repository cache. This will install the latest stable of NodeJS and NPM. We know it will get the latest version available by using `state=latest` when installing the `nodejs` package.

Vault

We often need to store sensitive data in our Ansible templates, Files or Variable files; It unfortunately cannot always be avoided. Ansible has a solution for this called Ansible Vault.

Vault allows you to encrypt any Yaml file, which typically boil down to our Variable files. Vault will not encrypt Files and Templates.

When creating an encrypted file, you'll be asked a password which you must use to edit the file later and when calling the Roles or Playbooks.

For example we can create a new Variable file:

```
1  ansible-vault create vars/main.yml
2  Vault Password:
```

After entering in the encryption password, the file will be opened in your default editor, usually Vim.

The editor used is defined by the `EDITOR` environmental variable. The default is usually Vim. If you are not a Vim user, you can change it quickly by setting the environmental variables:

Setting the editor used by Ansible Vault to Nano.

```
export EDITOR=nano
ansible-vault edit vars/main.yml
```

The editor can be set in the users profile/bash configuration, usually found at ~/.profile, ~/.bashrc, ~/.zshrc or similar, depending on the shell and Linux distribution used.

Ansible Vault itself is fairly self-explanatory. Here are the commands you can use:

Ansible-vault command options

```
$ ansible-vault -h
Usage: ansible-vault [create|decrypt|edit|encrypt|rekey] \
      [--help] [options] file_name

Options:
  -h, --help  show this help message and exit
```

For the most part, we'll use `ansible-vault create|edit /path/to/file.yml`. Here, however, are all of the available commands:

- **create** - Create a new file and encrypt it
- **decrypt** - Create a plaintext file from an encrypted file
- **edit** - Edit an already-existing encrypted file
- **encrypt** - Encrypt an existing plain-text file
- **rekey** - Set a new password on a encrypted file

Example: Users

I use Vault when creating new users. In a User Role, you can set a Variable file with users' passwords and a public key to add to the users' authorized_keys file (thus giving you SSH access).

Public SSH keys are technically safe for the general public to see - all someone can do with them is allow you access to their own servers. Public keys are intentionally useless for gaining access to a system without the paired private key, which we are not putting into this Role.

Here's an example variable file which can be created and encrypt with Vault. While editing it, it's of course in plain-text:

Editing encrypted file vars/main.yml

```
admin_password: $6$lpQ1DqjZQ25gq9YW$mHZAmGhFpPVVv0JCYUFaDovu8u5EqvQi.Ih
deploy_password: $6$edOqVumZrYW9$d5zj1Ok/G80DrnckixhkQDpXl0fACDfNx2EHnC
common_public_key: ssh-rsa ALongSSHPublicKeyHere
```

Note that the passwords for the users are also hashed. You can read Ansible's documentation on generating encrypted passwords[123], which the User module requires to set a user password. As a quick primer, it looks like this:

Using the mkpasswd command with SHA-512 encryption algorithm

```
# The whois package makes the mkpasswd
# command available on Ubuntu
$ sudo apt-get install -y whois

# Create a password hash
$ mkpasswd --method=SHA-512
Password:
```

This will generate a hashed password for you to use with the `user` module.

Once you have set the user passwords and added the public key into the Variables file, we can make a Task to use these encrypted variables:

File: tasks/main.yml

```
---
- name: Create Admin User
  user:
    name=admin
    password={{ admin_password }}
    groups=sudo
    append=yes
    shell=/bin/bash

- name: Add Admin Authorized Key
  authorized_key:
    user=admin
    key="{{ common_public_key }}"
    state=present

```

[123] http://docs.ansible.com/faq.html#how-do-i-generate-crypted-passwords-for-the-user-module

```
- name: Create Deploy User
  user:
    name=deploy
    password={{ deploy_password }}
    groups=www-data
    append=yes
    shell=/bin/bash

- name: Add Deployer Authorized Key
  authorized_key:
    user=deploy
    key="{{ common_public_key }}"
    state=present
```

These Tasks use the `user` module to create new users, passing in the passwords set in the Variable file.

It also uses the `authorized_key` module to add the SSH public key as an authorized SSH key in the server for each user.

Variables are used like usual within the Tasks file. However, in order to run this Role, we'll need to tell Ansible to ask for the Vault password so it can unencrypt the variables.

Let's setup a `provision.yml` Playbook file to call our `user` Role:

A Playbook calling the User Role

```
---
- hosts: all
  sudo: yes
  roles:
    - user
```

To run this Playbook, we need to tell Ansible to ask for the Vault password, as we're running a Role which contains an encrypted file:

Calling the provision.yml Playbook, which uses the User Role

```
ansible-playbook --ask-vault-pass provision.yml
```

You now have all the tools you need to begin using Ansible for development and production systems!

Ansible is capable of much more. If you're curious, check out how you can:

- Use Ansible for application deployment
- Use Ansible for rolling updates of infrastructure or applications
- Use Ansible with Continuous Integration (and/or Continuous Deployment) services to perform
- Use Ansible with Vagrant for development
- Ask Ansible to prompt you for variables
- Add public Roles to Ansible Galaxy
- Use Tower's free tier to get Ansible's GUI, useful for server configuration management in the browser

SSH

We use SSH to log into our servers, but it actually has a lot of other neat uses as well!

Logging in

Of course, we can use SSH to login to a server:

```
ssh user@hostname
```

If needed, we can specify a different port:

```
ssh -p 2222 user@hostname
```

Sometimes, if we have a lot of SSH keys in our `~/.ssh` directory, we'll often find that SSHing into servers with the intent of using a password results in a "too many authentication attempts" error. If we need to log into a server with a password, we can attempt to force password-based login. The following will stop SSH from attempting to use your SSH keys first, falling back to password-based authentication:

```
ssh -o "PubkeyAuthentication no" username@hostname
```

If you use AWS, and in other cases, you might get an id file such as a PEM file. In this case, you'll need to specify the specific identity file to use when logging in. We can do this with the `-i` flag:

```
ssh -i /path/to/identity.pem username@hostname
```

You may need to set your permissions on the pem file so only the owner can read/write/execute it: `chmod 0600 identity.pem`.

SSH Config

Configuring your local SSH config file is a very efficient way of using SSH.

If you want to setup aliases for servers you access often, you can create or edit the `~/.ssh/config` file and specify each servers you want to log into, along with the authentication methods to use.

Here are some examples you may add into your `config` file:

File: ~/.ssh/config

```
Host somealias
    HostName example.com
    Port 2222
    User someuser
    IdentityFile  ~/.ssh/id_example
    IdentitiesOnly yes

Host anotheralias
    HostName 192.168.33.10
    User anotheruser
    PubkeyAuthentication no

Host aws
    HostName some.address.ec2.aws.com
    User awsuser
    IdentityFile  ~/.ssh/aws_identity.pem
    IdentitiesOnly yes

Host somehostname anotherhostname athirdhostname
    HostName someserver.example.com
    User sharedusername
    IdentityFile  ~/.ssh/id_shared
    IdentitiesOnly yes
```

Logging into a server using a defined host ("alias") then becomes as easy as this:

```
ssh somealias
```

Note that we can define multiple hosts per definition as well!

Let's cover some of the options used above:

- **HostName** - The remote server host (domain or ipaddress) to connect to
- **Port** - The port to use when connecting
- **User** - The username to log in with
- **IdentityFile** - The SSH key identity to use to log in with, if using SSH key access
- **IdentitiesOnly** - "Yes" to specify only attempting to log in via SSH key (don't use password authentication)
- **PubkeyAuthentication** - "No" to specify you wish to bypass attempting SSH key authentication, defaulting back to password-based authentication

SSH Tunneling

SSH can be used for tunneling, which is essentially port forwarding. There's a few ways we can do this - Local (Outbound), Remote (Inbound), and some others (Dynamic and Agent Forwarding).

Some uses of this are to allow users to connect to remote services not listening on public networks, view your sites on your local machine or get around proxy restrictions, such as country-based limits.

Local Port Forwarding

Local port forwarding is what you use when you need to tunnel "through" a server's firewall or other limitation.

A common example is attempting to connect to a remote database which is either behind a firewall or is only listening to local connections.

For example, MySQL only listens to localhost connections by default. You can't remotely connect to it without editing MySQL's `my.cnf` configuration file and have it listen on a public network interface. There may also be a firewall preventing you from connecting to MySQL's port `3306` as well.

This is a common case when you are running MySQL on a server but want to connect to it from your computer's MySQL client, such as MySQL Workbench, Navicat, SequelPro or the command line MySQL client.

For this example a "remote" server means any computer that isn't yours, which **includes** virtual machines (guests) running inside of your host computer.

Assuming we have SSH access to the remote server, we can get around these access issues by creating a tunnel into the server. That looks like the following:

SSH tunneling - local port forwarding

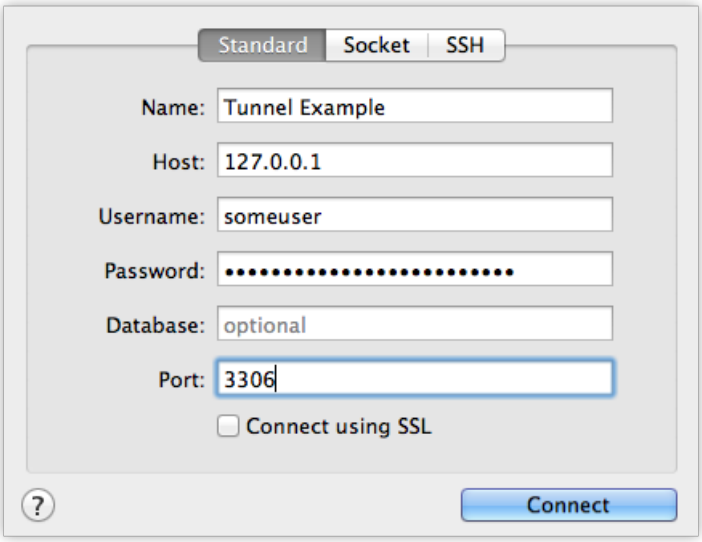
```
1 ssh -L 3306:localhost:3306 username@hostname
```

Let's go over this command:

- `-L` - Setup local port forwarding
- `3306` - The local port to forward

- `localhost:3306` - Within the remote server, what address and port to forward traffic to. Since the MySQL server is on the remote server, we're tunneling to the remote server's "localhost" on port 3306, which MySQL is listening to.
- `username@localhost` - The SSH username and host to connect to

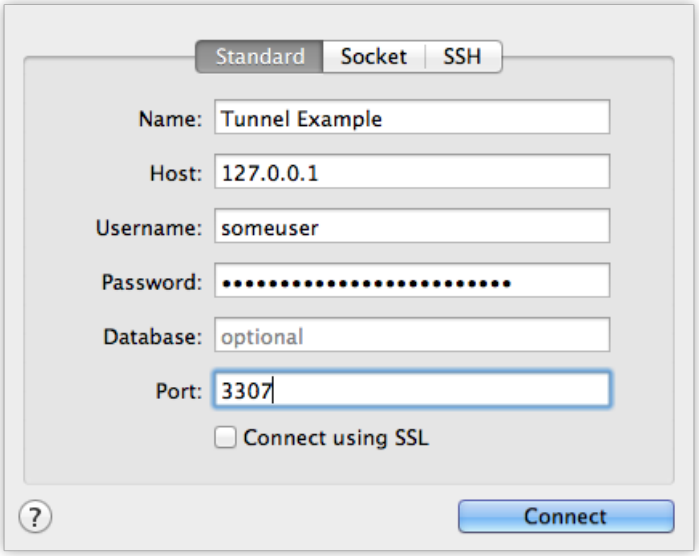
I can then use my local MySQL client to connect to the remote server as if it's a local one:



I used the same port locally and remotely, but I could have specified a different local port to use:

```
1 ssh -L 3307:localhost:3306 username@hostname
```

Then my local mysql client would have to connect to port `3307`, which would still tunnel to the remote server's local `3306`:



Remote Port Forwarding

Remote Port Forwarding is useful when you need to share your local computer with others who are outside of your network. One common use is to share your localhost web server with the outside world. This is how tools such as ngrok, pagekite, localtunnel and other "localhost tunneling" services work.

To accomplish this ourselves, we need a remote server all parties (our local computers and who we want to share with) can reach. Something like an AWS or Digital Ocean server will do.

Let's pretend our local computer has a web server running on port `8001`:

Local machine has a web server listening at port 8001

```
# On our local machine:
$ curl localhost:8001
Hi!
```

We want our friends to see our website, which simply says "Hi!". Let's use a remote server to forward requests to our local computer:

```
# Still on our local machine:
ssh -R 9000:localhost:8001 username@hostname
```

Let's go over this command:

- `-R` - Using remote port forwarding
- `9000` - The remote server's port to use (not our local server this time!)
- `localhost:8001` - The local address to forward to. Since our webserver is on localhost port `8001`, that's what we specify here. (the order of those arguments changed for -R over -L!)
- `username@hostname` - SSH access to the *remote* server

If our remote server's IP address was `123.123.123.123`, then our friends can access our website at `123.123.123.123:9000`, which will forward to our local site at `localhost:8001`!

To accomplish this, your remote server's firewall must not block port `9000`. You may also need to edit `/etc/ssh/sshd_config` and set the `GatewayPorts` directive to `yes`. (Don't forget to restart SSH after any changes to `sshd_config`).

One-Off Commands & Multiple Servers

You can run commands remotely using SSH without having to start a new terminal session and manually running commands.

Using the following "trick", you're connecting via SSH, running a command, and seeing the output all in one shot.

Let's run some simple commands on a remote server to see this in action. The following will run the `pwd` command. We'll see that it returns the default folder that we would be in when logging in. Then we'll run the `ls` command to see the directory's output:

```
# Run `pwd` command
$ ssh username@hostname pwd
/home/username

# Run `ls -la` command
$ ssh username@hostname ls -la
drwxr-xr-x 8 username username   4096 Jun 30 17:49 .
drwxr-xr-x 4 root     root       4096 Apr 28  2013 ..
-rw------- 1 username username  18589 Jun 30 17:49 .bash_history
-rw-r--r-- 1 username username    220 Apr 28  2013 .bash_logout
-rw-r--r-- 1 username username   3486 Apr 28  2013 .bashrc
-rw-r--r-- 1 username username    675 Apr 28  2013 .profile
drwxrwxr-x 2 username username   4096 Mar 15 14:21 .ssh
```

This lets us use SSH as a quick and easy way to check server statuses or perform quick operations. This can be used in scripts to automate running commands in multiple servers as well.

Basic Ansible

Using SSH in this manner is actually the basis of how the server provisioning tool Ansible works. It will run commands over SSH on groups of servers (in series or in parallel).

Let's see how that works. Note that we'll cover Ansible more in depth in the Server Configuration Management section of the book.

Start by installing Ansible on a local computer or server that will be doing the provisioning (usually not the server being provisioned):

```
sudo apt-add-repository ppa:ansible/ansible
sudo apt-get update
sudo apt-get install -y ansible
```

Ansible is "agentless", meaning it doesn't need to be running a service on the server it is provisioning. It works almost exclusively through SSH connections. We can provision servers from any server that can connect to other servers over SSH and has Ansible installed.

Next, configure one or more servers in the `/etc/ansible/hosts` directory:

```
[web]
192.168.22.10
192.168.22.11
192.168.22.12
```

This defines a "web" group of servers. I happen to have tested this with three local virtual machines, and so the addresses I put here are the three Ip addresses of my VMs. These can be IP addresses or host names.

Once that file is saved, we can run a command on all three servers at once!

```
ansible -k all -m ping -u vagrant
```

This will run "ping" on each server. You'll get some JSON output saying if they were successful or not.

The flags of that command:

- `-k` - Ask for password
- `all` - All servers configured in `/etc/ansible/hosts`. We could have specified the "web" group as well, which contained all of our defined servers
- `-m ping` - Use the ping module, which just runs the command "ping"
- `-u vagrant` - Login with user "vagrant", which will work if the hosts defined are other vagrant servers. Change the username as needed. It defaults to the username of the user running the command.

That's useful for running a simple command across all defined servers. More interestingly, you can run *any* arbitrary command using the "shell" module:

```
1 ansible -K all -m shell -u vagrant -a "apt-get install nginx"
```

Here, the `-a "apt-get install nginx` defines the command to run using the "shell" module.

I've also used `-K` over `-k` (uppercase vs lowercase). Uppercase "K" will use sudo with the command, and ask for the user's password.

More information on running ad-hoc commands with Ansible[124] can be found in the official documentation.

As mentioned, we'll cover Ansible more in depth in the Server Configuration Management section of the book. That will include an explanation of why the "shell" module may not be the best way to use Ansible.

[124] http://docs.ansible.com/intro_adhoc.html

Monitoring Processes

As some point you'll likely find yourself writing a script which needs to run all the time - a "long running process". These are scripts that should continue to run even if there's an error and should should restart when the system reboots.

These can be simple scripts or full-fledged applications.

To ensure our processes are always running, we need something to watch them. Such tools are Process Watchers. They monitor processes and restart them if they fail (usually due to unhandled errors or configuration issues), and ensure they (re)start on system boot.

A Sample Script

Linux distributions typically come tools to watch over processes. These tools are typically either Upstart or Systemd (altho the older SysV is still commonly used, often in conjunction with Upstart).

Most things we install with a package manager come with mechanisms in place for process watching using Upstart or Systemd. For example, when we install PHP5-FPM, Apache and Nginx with our package managers, they integrate with such systems so that they are actively monitored, leaving them much less likely to fail without notice.

Configuration for SysV/Upstart and Systemd isn't necessarily complex, but it's common to find that we can use some other solutions which might be more featured or easier to configure.

We'll cover a few of these monitoring tools. However, let's start with an example script that will serve as an example process to be monitored.

NodeJS script found at /srv/http.js

```
#!/usr/bin/env node
var http = require('http');

function serve(ip, port)
{
        http.createServer(function (req, res) {
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.write("\nSome Secrets:");
            res.write("\n"+process.env.SECRET_PASSPHRASE);
            res.write("\n"+process.env.SECRET_TWO);
            res.end("\nThere's no place like "+ip+":"+port+"\n");
        }).listen(port, ip);
        console.log('Server running at http://'+ip+':'+port+'/');
}

// Create a server listening on all networks
serve('0.0.0.0', 9000);
```

All this example service does is take a web request and print out a message. It's not useful in reality, but good for our purposes. We just want a service to run and monitor.

Note that the service prints out two environmental variables: "SECRET_PASSPHRASE" and "SECRET_TWO". We'll see how we can pass these into a watched process.

System Services

When Linux starts, the Kernel goes through a startup process, which includes initializing devices, mounting filesystems and then moves onto beginning the system init process.

The init process starts and monitors various services and processes. This includes core services such as the network, but also (usually) our installed applications such as Apache or Nginx.

There are various popular init processes. An old linux standard is System V Init (aka SysVinit or just SysV). A newer init process is Upstart. Finally there is Systemd.

Currently, Ubuntu has both SysVinit and Upstart installed and supported. They are often used in conjunction.

Debian has moved onto Systemd. Because Ubuntu is downstream from Debian, and after some internal turmoil, it will also include Systemd in a future release. Ubuntu 14.04 still uses Upstart/SysV.

In any case, all of these systems are responsible for managing processes in various stages of a system's life cycle: Start up, shutdown, reboot and during unexpected errors.

The serversforhackers.com[125] video site covers the following system init process monitors in a bit more depth.

System V Init (SysVinit, SysV)

You can tell your distribution is using SysVinit when you run commands such as `/etc/init.d/service-name [start|stop|restart|reload]`. Configurations for SysV are executable bash scripts found in the `/etc/init.d` directory. These scripts are responsible for handling the start, stop, restart and reload commands.

If you're interested to see what they look like or want to write your own, you can find a "skeleton" file. This is used as a baseline script found at `/etc/init.d/skeleton`. You can use it to copy and tweak as needed for your use.

SysVinit won't be covered futher here, but you can take a look to see which services have files in here.

[125] https://serversforhackers.com

Upstart

As mentioned, Upstart is the (relatively) newer system used by Ubuntu to handle process initialization and management. Configurations for Upstart are found in `/etc/init` rather than the `/etc/init.d` directory. Upstart configuration files end in the `.conf` extension.

Unlike SysVinit, the configurations in Upstart aren't directly executable scripts. Instead, the are configurations which follow Upstart's DSL (domain specific language).

An example configuration is as follows:

File: /etc/init/circus.conf

```
1  start on filesystem and net-device-up IFACE=lo
2  stop on runlevel [016]
3
4  respawn
5  exec /usr/local/bin/circusd /etc/circus/circusd.ini
```

This configuration for Circus (more on that tool later) will start Circus on boot, after the file system and localhost (lo) network have been initialized. It will stop at runlevel [016], essentially saying when the system shuts down (0), in single-user mode (1) or when the system reboots (6).

You can find more on Linux run levels in this IBM article[126].

The `respawn` directive will tell Upstart to respawn the process if it dies unexpectedly.

Finally the `exec` directive is a command used to run the process. Here we run the `circusd` process, passing it the `circusd.ini` configuration file.

Many programs try to run as daemons (in the background). Processes managed by Upstart generally should run in the foreground, allowing Upstart to monitor it.

That being said, Upstart *can* track certain processes which run as Daemons. See documentation on the use of the `expect` directive for more information.

Upstart uses the `initctl` command to control processes. We can run commands such as:

[126] http://www.ibm.com/developerworks/library/l-lpic1-v3-101-3/

```
# List available services
sudo initctl list

# Start and Stop Circus
sudo initctl start circus
sudo initctl stop circus

# Restart and Reload Circus
sudo initctl restart circus
sudo initctl reload circus

# Get the processes status (running or not running)
sudo initctl status circus
```

Ubuntu also has shortcuts for these - you can use the `start`, `stop`, `restart`, `reload` and `status` commands directly:

```
sudo start circus
sudo stop circus
sudo restart circus
sudo reload circus
sudo status circus
```

The Service Command

You may have noticed that everytime we've installed software, we've controlled it with the `service` command, such as the following:

```
sudo service apache2 start

sudo service nginx reload
```

Because Ubuntu (and other distributions) has transitioned between process monitors such as SysVinit and Upstart, the `service` command was created. This command is a common interface for many process monitors. For example, you can control SysV and Upstart processes using the same set of commands.

From the `service` man page:

> The SCRIPT parameter specifies a System V init script, located in /etc/init.d/SCRIPT, or the name of an upstart job in /etc/init. The existence of an upstart job of the same name as a script in /etc/init.d will cause the upstart job to take precedence over the init.d script.

The service command will check for the existence of a service by name in SysVinit's `/etc/init.d` and Upstart's `/etc/init`. If it finds a matching service in both, Upstart configurations will take precedence.

If you've ever wondered why you find tutorials using `/etc/init.d` and others using `service` to manage processes, now you know!

Systemd

Systemd is the newest init process manager. It's already used in many distributions (Fedora, RedHat 7, Debian 8, CoreOS, Arch and eventually Ubuntu).

Its use has been hotly contested. It's considered a "polyglot", taking over a lot of services such as logging, CRON and other system management. The "Linux Philosophy" has always been one of small tools that do one thing well. Whether Systemd ignores this philosophy is a topic of heated debate.

In any case, it seems to be winning in many distributions!

It's likely that Ubuntu will continue to use the `service` command even when Systemd is used, so our interface for managing processes can hopefully stay consistent.

Systemd uses the `systemctl` command to manage processes. Here are some examples of how to use it:

```
# Start/Stop services
sudo systemctl start some-service
sudo systemctl stop some-service

# Restart/Reload services
sudo systemctl restart some-service
sudo systemctl reload some-service

# Service status
sudo systemctl status some-service
```

As mentioned, Systemd also takes over other responsibilities, such as power management.

```
sudo systemctl reboot
sudo systemctl poweroff
sudo systemctl suspend
# And some others
```

Services in Systemd are called "units". Unit files (configuration for services) are located at `/etc/systemd/system` and contain the file extension `.service`.

Here's some example usage taken from CoreOS's example[127], which shows the starting of a Docker container:

```
[Unit]
Description=MyApp
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill busybox1
ExecStartPre=-/usr/bin/docker rm busybox1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name busybox1 busybox /bin/sh -c "while true; do\
 echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

First we define the **[unit]**, which has a description of simply "MyApp". Then we define `after` and `requires`, which defines that this service should only start after the `docker.service` is active.

Then we define the **[Service]** section. We disables the time check against how long a service should take to start, by setting `TimeoutStartSec` to `0`. Then there is a series of commands to execute before starting the service via the `ExecStartPre` directive. Finally, `ExecStart` defines the command to run.

Finally we define the **[Install]** section. The Install directive is used when `systemctl` enables or disables a service (but is ignored while Systemd is running a unit/service). Here we find the `WantedBy` directive, which defines the target which this service will be started with. Multi-user is sort of a catch-all target most commonly used.

Using These Systems

Most software we install will set up process monitoring automatically. If we install something with `apt-get` in this book, we likely will not have to do extra work to make the software start on system boot.

[127] https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd/

Software installed with language-specific package mangers (PIP, NPM, Gems) will need configuration to manage an application or process. This configuration will define how to handle start up, shutdown, reboot, and errors.

We'll use Upstart in some cases. In other cases, we'll use Upstart to monitor a third-party process monitor such as Supervisord or Circus.

For your own use, I suggest continuing to use Upstart for now. However, keep an eye on Systemd for when it becomes the defacto init system.

The remaining chapters of this section will cover some common "third-party" process monitors you can use.

Supervisord

Written in Python, Supervisord is a simple and extremely popular choice for process monitoring. Its excellent documentation is found at http://supervisord.org[128] Let's check out the package on Ubuntu:

```
$ apt-cache show supervisor
Package: supervisor
Priority: extra
Section: universe/admin
Installed-Size: 1485
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Qijiang Fan <fqj1994@gmail.com>
Architecture: all
Version: 3.0b2-1
Depends: python, python-meld3, python-pkg-resources (>= 0.6c7)
Filename: pool/universe/s/supervisor/supervisor_3.0b2-1_all.deb
Size: 313972
MD5sum: 1e5ee03933451a0f4fc9ff391404f292
SHA1: d9dc47366e99e77b6577a9a82abd538c4982c58e
SHA256: f83f89a439cc8de5f2a545edbf20506695e4b477c579a5824c063fbaf94127c1
Description: A system for controlling process state
Description-md5: b18ffbeaa3a697e8ccaee9cc104ec380
Homepage: http://supervisord.org/
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Origin: Ubuntu
```

We can see that we'll get version 3.0b2. That latest is version 3.1 (as of this writing), but 3.0b2 is good enough.

A Chain of Process Monitors

We can get a newer version of Supervisord by installing it manually or using Python's Pip. However then we'd lose out on the benefits that the APT package gives us. We'd have to make sure all the dependencies are met and setup Upstart or SysV ourselves.

Instead, we'll install Supervisord with APT. When we do, note that Supervisord is actually monitored by Upstart! There is a "chain" of monitoring.

[128] http://supervisord.org/

Upstart will monitor Supervisord, which in turn will monitor whatever we configure.

If you'd like, you can skip this chaining by using a system process monitor such as Upstart, SysV or Systemd.

The benefit of "third-party" process monitors such as Supervisord are extra features you may need or want.

Installation

To install Supervisord, we can simply run the following (note that its often referred to as "supervisor" instead of "supervisord"):

```
sudo apt-get install -y supervisor
```

Installing it as an APT package gives us the ability to treat it as a service (since Upstart/SysV is monitoring it!):

```
sudo service supervisor start
```

Configuration

Configuration for Supervisord is found in `/etc/supervisor`. If we look at the configuration file `/etc/supervisor/supervisord.conf`, we'll see at the following at the bottom:

```
[include]
files = /etc/supervisor/conf.d/*.conf
```

So, any files found in `/etc/supervisor/conf.d` and ending in `.conf` will be included. This is where we can add configurations for our services.

Now we need to tell Supervisord how to run and monitor our Node script. To do so, we'll create a configuration which tells Supervisord how to start and monitor the Node script.

Let's create a configuration for it called `webhook.conf`.

File: /etc/supervisor/conf.d/webhook.conf

```
[program:nodehook]
command=/usr/bin/node /srv/http.js
directory=/srv
autostart=true
autorestart=true
startretries=3
stdout_logfile=/var/log/webhook/nodehook.out.log
stderr_logfile=/var/log/webhook/nodehook.err.log
user=www-data
environment=SECRET_PASSPHRASE='this is secret',SECRET_TWO='another secret'
```

As usual, we need to go over the options set here:

program:nodehook

Defines the name of the program to monitor. We'll call it "nodehook" (the name is arbitrary).

command

Define the command to run. We use `node` to run the `http.js` file. If we needed to pass any command line arguments/flags, we could do so here.

directory

We can set a directory for Supervisord to "cd" into for before running the monitored process, useful for cases where the process assumes a directory structure relative to the location of the executed script.

autostart

Setting this "true" means the process will start when Supervisord starts (essentially on system boot). Because Supervisord itself will start on system boot, thanks to the configured Upstart/SysV, we know that our Node process will be started in turn after Supervisord.

autorestart

If this is "true", the process will be restarted if it exits unexpectedly.

startretries

The number of retries to attempt before the process is considered "failed".

stdout_logfile

The file to write any regular (stdout) output.

stderr_logfile

The file to write any error (stderr) output.

Note that we've specified some log files to be created inside of the `/var/log/webhook` directory. Supervisord won't create a directory for logs if they do not exit; We need to create them before running Supervisord:

```
sudo mkdir /var/log/webhook
```

user

The process will be run as the defined user

environment

Environment variables to pass to the process. You can specify multiple in a comma-separated list, such as `key1="value1",key2="value2",key3="value3"`. This is useful if your script needs to authenticate against other services such as an API or database.

Controlling Processes

Now that we've configured Supervisord to monitor our Node process, we can read the configuration in and then reload Supervisord, using the `supervisorctl` tool:

```
1 supervisorctl reread
2 supervisorctl update
```

Our Node process should be running now. We can check this by simply running `supervisorctl`:

```
$ supervisorctl
nodehook                 RUNNING    pid 444, uptime 0:02:45
```

You can exit the supervisorctl tool using ctrl+c.

We can double check this using the `ps` command:

```
$ ps aux | grep node
www-data   444  0.0  2.0 659620 10520 ?  Sl   00:57   0:00 /usr/bin/node \
    /srv/http.js
```

It's running! If we check our sample Node process listening at localhost:9000, we'll see the output generated which include the environment variables.

```
$ curl localhost:9000
Some Secrets:
this is secret
another secret
There's no place like 127.0.0.1:9000
```

If your process is not running, try explicitly telling Supervisord to start process "nodehook" via `supervisorctl start nodehook`

There are other things we can do with the `supervisorctl` command as well. Enter the controlling tool using `supervisorctl`:

```
$ sudo supervisorctl
nodehook     RUNNING    pid 444, uptime 0:15:42
```

Then you can use the `help` command to see available commands:

```
supervisor> help
default commands (type help <topic>):
=====================================
add   clear fg       open quit   remove restart  start  stop update
avail exit  maintail pid  reload reread shutdown status tail version
```

We can try some more commands. Let's stop the `nodehook` process:

```
supervisor> stop nodehook
nodehook: stopped
```

Then we can start it back up:

```
supervisor> start nodehook
nodehook: started
```

Use ctrl+c or type "exit" to get out of the supervisor tool.

Those commands can also be run directly, without being "in" the supervisorctl tool:

```
supervisorctl stop nodebook
supervisorctl start nodebook
```

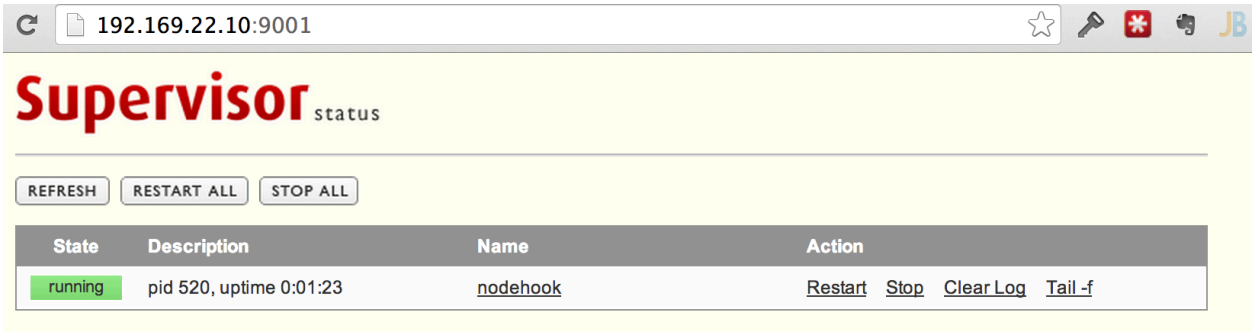
Web Interface

We can configure the web interface that comes with Supervisord. This lets us see a list of all monitored processes, as well as take action on them (restarting, stoppping, clearing logs and checking output).

Inside of `/etc/supervisord.conf`, add this:

```
[inet_http_server]
port = 9001
username = user # Basic auth username
password = pass # Basic auth password
```

If we access our server in a web browser at port 9001, we'll see the web interface after entering in the basic auth username and password:



Clicking into the process name ("nodehook" in this case) will show the logs for that process.

If you make use of this interface, you'll want to ensure that it's not publicly available, usually accomplished using the firewall.

Forever

In the Node world, Forever is a popular choice for process watchers. If you already have Node and NPM on your server, it's very easy to use! Its documentation is found at the GitHub project page **nodejitsu/forever**.

One caveat to Forever is that it's not meant to persist processes across a system (re)boot and doesn't necessarily handle graceful restarts. This limits its usefulness a bit, but it's very easy to use!

However, Forever can watch for file changes, making it a nice development tool.

Installation

To install Forever, we'll use NPM, the Node Package Manager. Forever is typically installed globally, so we'll use "sudo" and the `-g` flag.

```
1  sudo npm install -g forever
```

This assumes that Node and NPM is already installed on your system.

Usage

There's no configuration files for Forever - we can just start using it.

Let's see an example of using Forever to run our `/srv/http.js` script:

```
1  sudo forever start -l /var/log/forever/forever.log \
2       -a -o /var/log/webhook/out.log -e /var/log/webhook/error.log \
3       --sourceDir /srv http.js
```

There's a bunch of options (and more in the docs). Let's cover the flags used above:

- **start** - We're telling Forever to start a new process. There are other actions forever can take, such as listing each process, stopping all processes, restarting all processes, checking logs, and more.

- **-l /var/log/forever/forever.log** - Specify the log used for Forever's output.
- **-a** - Tell Forever to append to the log files specified, instead of overwrite them with new log output.
- **-o /var/log/webhook/out.log** - Where to log regular output from the process being watched.
- **-e /var/log/webhook/error.log** - Where to log error output from the process being watched.
- **–sourceDir /srv** - What directory to run the process relative to
- **http.js** - the script to run. We don't specify the full path `/srv/http.js` since the `--sourceDir` option will fill in the file path for us.

You may need to create the log directories, for example `/var/log/forever` and `/var/log/webhook` in this example.

Not shown here was the `-c` option, which can use if we're not running a node script. For example if we're running a bash script, we could use `-c /bin/bash`.

```
root@49ddc9d98b94:/# sudo forever start --sourceDir=/srv/ -l /var/log/forever/forever.log -a -o /var/log/webhook/out.log -e /var/log/webhook/error.log http.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
info:    Forever processing file: http.js
root@49ddc9d98b94:/# sudo forever list
info:    Forever processes running
data:        uid  command         script       forever pid logfile                 uptime
data:    [0] RX8F /usr/bin/nodejs /srv/http.js 149     155 /var/log/forever/forever.log 0:0:0:49.659
root@49ddc9d98b94:/#
```

Circus

Circus is a more fully featured process manager. Similar to Supervisord, it's written in Python but doesn't require any knowledge of Python for its use, with the minor exception of possibly using a Python package manager to install it.

Installation

Circus is available to install via Python's package manager Pip. Pip will manage Python dependencies, but not necessarily other system dependencies, so external libraries used by the Circus Python package will need to get installed separately.

Circus uses ZeroMQ[129] for messaging between system events and to send commands. It also prefers the use of Python's Virtualenv, which is similar to rbenv in the Ruby world. Virtualenv let's you install and use Python in its own environment, allowing the use of different versions of Python and Python libraries within each environment.

On Ubuntu, we can install the system dependencies (ZeroMQ/LibEvent, Python Dev, Virtualenv and Python Pip) like so:

```
sudo apt-get install -y libzmq-dev libevent-dev python-dev python-virtualenv
```

Then, to install Circus, we can use Virtualenv to setup an environment and Pip to install Circus:

```
# Create directory for Circus
# and change owner to current user
sudo mkdir /var/opt/circus
sudo chown ($whoami) circus

# Setup virtual environment for Python
virtualenv /var/opt/circus

# Install Circus & Related
cd /var/opt/circus
./bin/pip install circus
./bin/pip install circus-web
```

[129] http://zeromq.org/

Once we setup a virtual environment via the `virtualenv` command, we used the environment's version of Pip to install Circus and Circus-Web.

If you log out and back into your server, the environment setup with Virtualenv will need to be re-initiated. You can do that by sourcing the "activate" file created within each environment:

```
source /var/opt/circus/bin/activate
```

More information on virtualenv can be found in the Virtualenv docs[130].

Once Circus and Circus-web are installed, we can begin using Circus to monitor our sample NodeJS process.

Configuration

Circus uses `.ini` files for configuration. We'll create a new configuration file for our NodeJS script called `webhook.ini`:

File: /var/opt/circus/webhook.ini

```
[circus]
statsd = 1
httpd = 1
httpd_host = 127.0.0.1
httpd_port = 9002

[watcher:webhook]
cmd = /usr/bin/nodejs /srv/http.js
numprocesses = 1
max_retry = 3
stdout_stream.class = FileStream
stdout_stream.filename = ./webhook.out.log
stderr_stream.class = FileStream
stderr_stream.filename = ./webhook.err.log

[env:webhook]
SECRET_PASSPHRASE = some secret
SECRET_TWO = another secret
```

There's a lot happening here, let's cover it.

[130] http://docs.python-guide.org/en/latest/dev/virtualenvs/

circus

This is the section for the configuration of Circus itself, rather than being something specific to our NodeJS process.

Here we enable the web interface for Circus (Circus-Web):

- **statsd** - This enables the stats module, which can read system resource usage of Circus and its monitored processes
- **httpd** - Enabling this tells Circus-Web to start the circushttpd daemon, which is its web interface
- **httpd_host** and **httpd_port** - Set the host and port to bind the circushttpd daemon. This defaults to `localhost:8080` if not specified.

watcher:webhook

Here we define a watcher and name it "webhook".

The `cmd` we've set is simply to have nodejs run our `http.js` file defined in the beginning of this chapter.

The `numprocesses` is set to 1, as Node scripts run as a single processes. This is not to say that Circus can't run multiple instances of our Node script - in fact it can. However we'll just run one instance of the `http.js` script.

The `numprocesses` directive has some interesting implications. Circus can actually control and monitor multiple processes for us. We can use Circus to "spin up" multiple processes of an application.

For example, if Circus is monitoring multiple processes of an application. This is similar to setting the number of processes that Apache or PHP-FPM would use, if they did not control that themselves.

We set the `max_retry` to three - Circus will try a max of three times to restart the process if it dies.

Next we'll define our log files. We need to set the `stdout_stream.class` to `FileStream`, which will write to a file. Then we set the `stdout_stream.filename` to the log file. In this case, I just set it as a file in the same directory, however you may want it saved somewhere in `/var/log`.

We do the same for our error log, by defining the `stderr_*` directives rather than the `stdout_*` directives.

There are other available options for log files, such as setting the output format and handling the rotation of logs. You can find them in the official documentation[131].

[131] http://circus.readthedocs.org/en/0.11.1/

User and Group

We can set what user and group to run the process as via the `uid` and `gid` parameters. These directives expect user and group ID numbers instead of the user/group names.

You can find your user's uid and gid by simply typing the `id` command:

```
# Typing in "id" as user "vagrant"
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

You can specify a username to get any user's information as well:

```
$ id www-data
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

We can use uid/gid of 33 if we wanted our process to run as www-data. By default, Circus will run processes as the same user/group that Circus is run as.

Note that we didn't use Circus with "sudo" - it's not run as root in these examples. This is usually done as a security measure. You can run it with sudo in conjunction with using the uid/gid options (similar to how we did with Supervisord). Be aware that running circusd as root exposes you to potential privilege escalation bugs (vulnerabilities).

Circus has more security information found in the security section[132] of the documentation.

env:webhook

Our NodeJS script looks for two environmental variables `SECRET_PASSPHRASE` and `SECRET_TWO`. Circus allows us to pass environment variables to the script to use as well. Here we can set simply key and value pairs for our script to use.

Controlling Processes

Circus comes with a `circusctl` command which we can use to control monitored processes. This is very similar to Supervisord's `supervisorctl`.

We can run one-off commands, or we can enter the controller:

[132] http://circus.readthedocs.org/en/latest/design/security/

Enter the Circus controller

```
./bin/circusctl
```

Run the "help" command to see all the available commands:

```
(circusctl) help
Documented commands (type help <topic>):
========================================
add     globaloptions  list          numwatchers  reloadconfig  signal  stop
decr    help           listen        options      restart       start
dstats  incr           listsockets   quit         rm            stats
get     ipython        numprocesses  reload       set           status
```

If we use `list`, we'll see a list of three processes, Circus's own running processes and the webhook:

- circusd-stats - The stats module we enabled
- circushttpd - The httpd module (web interface)
- webhook - The NodeJS script we are monitoring

Use ctrl+c to exit `circusctl`. We can also use one-off commands without entering the controller:

Running `./bin/circusctl stop` will stop all processes. We can define a specific process as well - running `./bin/circusctl stop webhook` will stop NodeJS script. (We can run `start webhook` to restart it).

The `reloadconfig` config option will re-read configuration if we change the `webhook.ini` file. Then we can run `reload` to make the changes take effect:

```
./bin/circusctl reloadconfig
./bin/circusctl reload
```

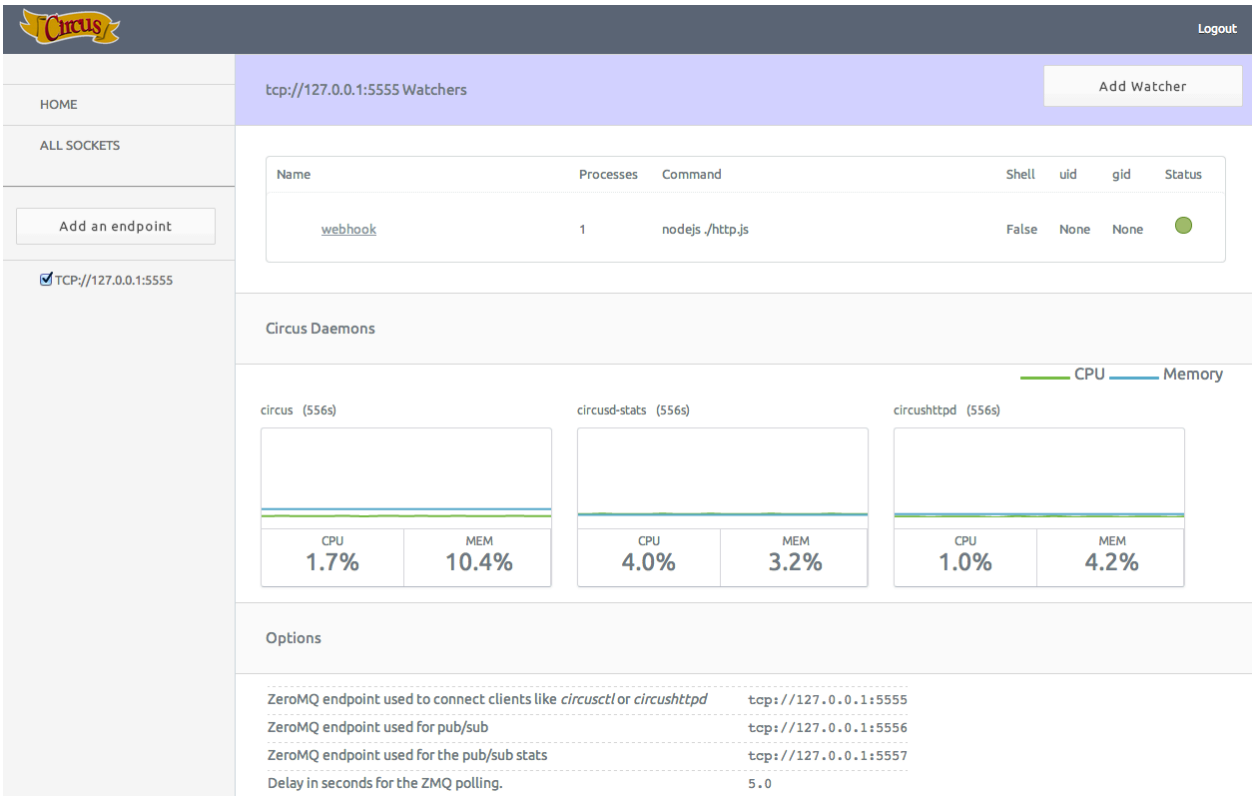
Interestingly, circusctl is "just" a ZeroMQ arbiter - it's just sending commands in the form of messages, acting as a ZeroMQ client. You can actually send your own commands programmatically. For example, the JSON to stop all processes looks like this:

```
{
    "command": "stop",
    "properties": {
        "waiting": False
    }
}
```

The Circus web interface will tell you what socket (IP + Port) to send ZeroMQ messages to, for example `tcp://127.0.0.1:5555`.

Web Interface

As mentioned, Circus has a web interface. This makes use of Socket.io to send "real-time" updates via the Stats module which we enabled alongside the web console (httpd).



This interface shows the processes being monitored and even lets you add additional processes to a watched process, if it supports it. For example, if Circus is monitoring a Python application, it can add more application listening processes. This is similar to how we can configure more processes in Apache or PHP-FPM.

Starting on Boot

When covering Supervisord, we mentioned that installing it via a package manager actually sets up an Upstart/SysV configuration, so that the system is monitoring Supervisord, while Supervisord was monitoring our NodeJS script.

Circus is in the same situation; It requires something to monitor it so that it starts on system boot and restarts if there's an unexpected error. Because we installed Circus using Pip, rather than a system package manager, there is no Upstart, SysV or Systemd configuration setup so ensure Circus is always running.

The Circus Deployment documentation[133] has information on how to create such a configuration. They include two examples to use which can handle monitoring Circus processes as well as starting them when the system boots. These are Upstart and Systemd.

Ubuntu comes with Upstart, so we'll concentrate on that here.

Ubuntu is moving on to use Systemd, but has not yet made the move.

To get Circus to start on system boot (and to restart it if Circus fails), we can create a Upstart configuration for Circus. All files inside of the `/etc/init` directory ending in `.conf` will be used by Upstart. We'll add our configuration for Circus there.

The documentation for Circus give us this Upstart configuration to use:

File: /etc/init/circus.conf

```
start on filesystem and net-device-up IFACE=lo
stop on runlevel [016]

respawn
exec /usr/local/bin/circusd /etc/circus/circusd.ini
```

This will start and stop Circus on boot, and respawn it if it stops expectantly. It will start Circus via the `exec` directive. However, the above file paths are wrong for our NodeJS example. Additionally, it assumes you aren't using Virtualenv (it doesn't source `/tmp/circus/bin/activate`).

We can adjust this script to take care of that. To do so, we'll use the `script` directive, which will allow us to do more than we could with the simple `exec` line:

[133] https://circus.readthedocs.org/en/latest/for-ops/deployment/

File: /etc/init/circus.conf

```
start on filesystem and net-device-up IFACE=lo
stop on runlevel [016]

respawn

script
    cd /tmp/circus
    . ./bin/activate
    ./bin/circusd ./webhook.ini
end script
```

This lets use put a shell script between `script` and `end script`. Here we `cd` into the circus directory. Then we source the `bin/activate` file (using the `.` notation rather than the command `source`, which we can't use in this context). Finally we run `circusd`, passing it the `webhook.ini` configuration.

Note that we didn't run Circus as a daemon (via the `--daemon` flag). Upstart will run it for us, monitoring the circus process and keeping it alive itself.

Once the `/etc/init/circus.conf` file is created, we can start using it with Upstart's commands:

```
# Check the Upstart script exists
$ sudo initctl list | grep circus
circus stop/waiting

# Check the status of Circus
$ sudo status circus
circus stop/waiting

# Start Circus
$ sudo start circus

# Stop Circus
$ sudo start circus
```

So we can control Circus via Upstart, and know that it will restart along with the server.

Development and Servers

Many people work on Macintoshes or Linux servers. These usually come with the ability to serve static content out of the box, and there are even some simple options to get fancier with dynamic content. Here are some examples of some useful tools which may be hiding under your nose.

Serving Static Content

Built-In

Your Mac has a super-easy way to server static content out of the box, without installing *anything*. This makes use of the fact that Mac's come with Python, and Python's standard library contains the super-handy SimpleHTTPServer module.

Serving static files using Python

```
cd /path/to/static/html
python -m SimpleHTTPServer 8000
```

After running the above command, you'll see something like `Serving HTTP on 0.0.0.0 port 8000 ...` - you're good to go! Head over to http://localhost:8000 in your browser to see what you find!

The beauty of this is that you can run this from any directory/location on your Mac, even off of shared network drives - as long as your Mac can read the files.

Serving static files from a mounted network drive

```
cd /Volumes/SomeNetworkDrive/path/to/html
python -m SimpleHTTPServer  # awww, yeah
```

Mac's system-install Python also comes bundled with Twisted, another Python web server! You can run this (supposedly production-grade) static file server using Twisted with this command:

```
twistd -n web -p 8888 --path /path/to/html
```

This isn't limited to Python; You can do this with the system-installed Ruby as well:

Serving static content with Ruby

```
ruby -run -e httpd /path/to/html -p 8888
```

NodeJS

If you have NodeJS installed, you can find an equally simple static file server.

NodeJS script to serve static content, via https://gist.github.com/rpflorence/701407

```
var http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs")
    port = process.argv[2] || 8888;

http.createServer(function(request, response) {

  var uri = url.parse(request.url).pathname
    , filename = path.join(process.cwd(), uri);

  path.exists(filename, function(exists) {
    if(!exists) {
      response.writeHead(404, {"Content-Type": "text/plain"});
      response.write("404 Not Found\n");
      response.end();
      return;
    }

    if (fs.statSync(filename).isDirectory()) filename += '/index.html';

    fs.readFile(filename, "binary", function(err, file) {
      if(err) {
        response.writeHead(500, {"Content-Type": "text/plain"});
        response.write(err + "\n");
        response.end();
        return;
      }

      response.writeHead(200);
      response.write(file, "binary");
      response.end();
    });
  });
}).listen(parseInt(port, 10));

console.log("Static file server running at\n  => http://localhost:" + port + "/\\
nCTRL + C to shutdown");
```

You can place this anywhere and then use it to serve files from the location of the NodeJS script:

```
# Run NodeJS static server from directory
# containing the static files
node static_server.js 8888
```

Dynamic Content

Serving dynamic content is, of course, more complex.

If you are on Mac's Mavericks, you actually have PHP 5.4+ installed. This means PHP's built-in web server will work! This will serve static files and process PHP files.

```
cd /path/to/php/files
php -S localhost:8888
```

Of course, if your PHP application requires them, you'll need to install modules such as mcrypt, PDO, GD or other PHP modules which might not come with Mac OS. You can use the Brew package manager to easily install these dependencies.

However, consider using a virtual machine (perhaps with Vagrant) to more easily be able to install and manage application dependencies, as well as to keep your Macintosh clean of such things. Avoiding the pain of configuring "server stuff" on your Macintosh is worth it!