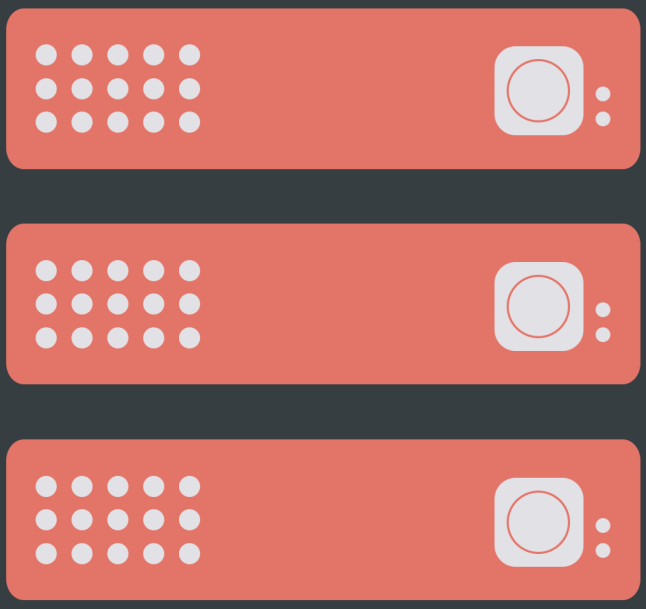


Case Studies

Servers for Hackers

Chris Fidao

Servers for Hackers Case Studies

Case Studies for the Servers for Hackers Book

Chris Fidao

Contents

Server Quick Start Guide

Here we'll cover how I spend the first 15 minutes in any server I build. This includes setting up users, security and installing basic software.

Basic Software

When we get our hands on a new server, we should install some basic tools which we'll use again and again. What tools these are will change for you as dictated by requirements and what tools you become comfortable with.

Here's what we'll 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" we saw in the chapter on Vagrant)
- **vim** - The timeless editor. 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** - Unlike the others above, this is specific to Ubuntu. It is used to manage the repositories that you install software from. We'll use this to get more current versions of some of our software.

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

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

Users and SSH Access

Here's our next steps:

1. Create a new (non-root) user

2. Allow this user to use "sudo" for administrative privileges
3. Stop user "root" from remotely logging in via SSH
4. Configure SSH to change the port and add other restrictions
5. Create an SSH key on our local computer new our new user
6. Turn off password-based authentication on our server altogether, 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 a 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:

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

Once you're logged in, you can simply use the `adduser` command to create a new user:

```
1 sudo adduser someusername
```

This will ask you some information, most importantly your password. You can leave the other fields blank. Take the time to add a somewhat lengthy, secure password. Keep in mind you may be asked for your password to run privileged commands down the line.

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" in order to run commands as root. How easily you can do this changes per operating system.

On Ubuntu, you can simply add the user to the pre-existing "sudo" group.

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

Let's go over that command:

- **usermod** - Command to modify and 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".

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 has access but doesn't know the 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.

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.

Configure SSH

Many automated bots are out there sniffing servers for vulnerabilities. One common thing checked is whether the default SSH port is open for connections. Because this is such a common attack vector, it’s often recommend that you change the SSH port away from the default **22**.

We’re allowed to assign ports between 1025 and 65536, inclusive. To do so, you can simply change the Port option in the same `/etc/ssh/sshd_config` file:

File: /etc/ssh/sshd_config

```
Port 1234
```

This will no longer accept connections from the standard port 22. A side affect of this is needing to specify the port when you log in later.

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

# We need to add the -p flag to specify the port
ssh -p 1234 user@hostname
```

To get even more secure, we can also explicitly define a list of users and groups who are allowed to login. This is configurable with the `/etc/ssh/sshd_config` file.

I like to restrict this based on groups, using the `AllowedGroups` directive. This is useful for simplifying access - you can simply 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 will let users of group “sudo” and “canssh” SSH access. We’ve already assigned our new user the “sudo” group, so they are good to go.

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

```
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. 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.

First we need to create an SSH key on our local computer. Run this on your *local* computer, the one that will need access to your server:

```
# 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[1].
- **-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 key files created (id_myidentity and id_myidentity.pub in this case)

Creating an SSH key will ask you for a password! You can either leave this blank (for passwordless access) or enter in a password. I **highly** suggest using a password. This makes it so attackers require both your private key AND your SSH password to gain SSH access. This is in addition to your user's password needed to run any sudo commands on the server!

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 go into your server as your new user ("someusername" in our example). It's important to log in as the user you want to log in as (not root), so that the public key is added for that user.

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

```
# 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)
```

This is appending the public key from our local computer to the authorized_keys file of the newly created user on our server.

Once the authorized_keys file is saved, you should be able to login from your local computer 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. You'll need to enter in your password created while generating your SSH key, if you elected to use a password.

You may 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`

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 just a password.

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

```
# Use nano instead of nano 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 file, 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's true! Don't close your current or backup connection, just in case you run into issues.

Setting Up the Firewall

The firewall offers some really basic protections on your server - it's a very important tool.

The following is basic list of INPUT (inbound traffic) rules we'll be building:

```
target     prot opt in     out     source      destination
ACCEPT     all  --  lo     any     anywhere    anywhere
ACCEPT     all  --  any    any     anywhere    anywhere      ctstate RELATED,EST\
ABLISHED
ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:ssh
ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:http
ACCEPT     tcp  --  any    any     anywhere    anywhere      tcp dpt:https
DROP       all  --  any    any     anywhere    anywhere
```

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

1. Accept all traffic on "lo", the "loopback" interface[2]. This is essentially saying "Allow all **internal** traffic to pass through"
2. Accept all traffic from currently established (and related) connections. This is typically set so you don't accidentally block yourself from the server when in the middle of editing firewall rules
3. Accept TCP traffic (vs UDP[3]) over port 22 (which iptables labels "ssh" by default). If you changed the default SSH port, this will show the port number instead
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

Adding these rules

Run `sudo iptables -L -v` to list your current rules. If this is a new server, there are likely no rules, it will look like this:

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

[3] https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=what%20is%20udp

```
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
```

If you have some firewalls rules already, you should review the chapter on the Iptables firewall to see the best way to proceed in your case.

Assuming there are no rules established, we'll build the above rules by adding rules to the INPUT chain. The INPUT chain is the set of rules that is checked against for all inbound network traffic.

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

The above assumes the usual port 22 is used for your SSH port. Use whatever port you used to the SSH service, if you changed it from the default 22.

The above is good for a web server. The only incoming traffic allowed is SSH traffic and the standard web ports 80/443 (http/https). Open up more TCP or UDP ports as you need for other services which may need to connect to your server.

Saving Firewall Rules

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

On Ubuntu, we can use the `iptables-persistent` package to 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 a file that iptables-persistent will read from. This file will contain the output from the `iptables-save` command. It will load those rules in when the server boots!

```
sudo iptables-save > /etc/iptables/rules.v4
```

When that's done, restart iptables-persistent:

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

Fail2Ban

Fail2Ban will monitor for intrusion attempts on your server and use the iptables firewall. It will ban specific hosts (usually an IP address) if they meet a configured threshold.

When Fail2Ban bans a host, it will use the iptables firewall to block a host from which too many bad attempts (or other "bad behavior") came from.

Install & Configure Fail2Ban

Simply run this to install Fail2Ban:

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

Next we need to configure Fail2Ban. There's a configuration file called `jail.conf` which we can copy and edit to suit our needs. If we name the copy `jail.local`, Fail2Ban will automatically use it:

```
sudo cp /etc/Fail2Ban/jail.conf /etc/Fail2Ban/jail.local
```

Fail2Ban will, by default, monitor logs pertaining SSH and block hosts which have too many bad login attempts. This is enough for now - we can consider Fail2Ban configured and installed.

Just ensure we get our new configratioan file and Fail2Ban is started:

```
sudo service fail2ban reload
```

Automatic Security Updates

You may want your server to run automatic updates. In Ubuntu, we can specifically do automatic *security* updates. This gives us the ability to skip other non-essential updates.

Install

First, we'll ensure `unattended-upgrades`[4] is installed:

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

Configure

Then update `/etc/apt/apt.conf.d/50unattended-upgrades` (the number preceding the filename might vary a bit). Make sure `"Ubuntu trusty-security";` is uncommented, while the remaining "Allowed-Origins" listed are commented out:

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

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

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";
```

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

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

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 be run daily automatically, as all "Periodic" items are set to run via the daily cron. If you're curious, you can find that configured in the `/etc/cron.daily/apt` file.

All Set!

Once these are done, you're all set! We've done quite a lot! We've installed software, setup users and secured user access, setup our firewalls, setup some access monitoring and finally configured auto security updates.

Case Study: Nginx with Multiple Sites

Here's a fairly typical case. We have a server with Nginx, and we want to setup a few websites on it. For this case study, we'll look at the setup to host multiple sites - one static and one dynamic application.

This will setup:

- Installation and general configuration
- Configuration for static assets and static sites including:
  - Cache headers
  - Cache busting assets
  - Headers for IE compatibility
- Configuration for dynamic sites, including
  - Headers for CORS

Installing Nginx

The Nginx chapter goes into installing and configuration Nginx. We'll go through the boiler-plate of installation quickly. Finally we'll get some explanation on specific configurations used.

First, we need to intall Nginx on our server:

```
sudo add-apt-repository -y ppa:nginx/stable
sudo apt-get update
sudo apt-get install -y nginx
sudo service nginx start
sudo update-rc.d nginx defaults  # Set Nginx to start on boot
```

Ubuntu comes with a default site configured in `/etc/nginx/sites-available/default`. Here's what that looks like with the comments removed:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

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

    # Make site accessible from http://localhost/
    server_name localhost;

    location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }
}
```

This file is symlinked to `/etc/nginx/sites-enabled/default`, so we know it's currently enabled.

We're going to make new configurations for the static and dynamic sites we create.

We don't need the default site declaration. However, we won't delete it. Instead, the default site can instead be simplified down to the following:

File: /etc/nginx/sites-available/default

```
server {
  listen 80 default_server;

  # Only needed if you use ipv6:
  listen [::]:80 default_server ipv6only=on;

  return 444;
}
```

This returns HTTP status code 444, which Nginx uses to mean "No Response". This prevents a site from being served for HTTP requests missing a `Host` header.

Some configuration we'll see here is borrowed directly from H5BP's server-configs-nginx[5] repository. This repository has a great set of defaults for commonly forgotten needs, such as setting cache headers on static assets.

[5] https://github.com/h5bp/server-configs-nginx

Static Site

Let's make a configuration for the static site first. In this case, it will be the configuration for a static site: serversforhackers.com!

Let's create a new configuration for the site:

File: /etc/nginx/sites-available/serversforhackers

```
server {
    listen 80;

    server_name *.serversforhackers.com;
    return 301 $scheme://serversforhackers.com$request_uri;
}

server {
    listen 80;

    server_name serversforhackers.com;

    root /var/www/serversforhackers.com/public;
    index index.html index.htm;

    charset utf-8;

    access_log /var/log/nginx/serversforhackers.com.log;
    error_log  /var/log/nginx/serversforhackers.com-error.log error;

    location / {
        try_files $uri $uri/ =404;
    }

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

    include h5bp/basic.conf;
}
```

Most of these options are covered in the chapter on Nginx. The first `server` block will redirect any subdomain (such as `www`) to the root domain.

Use of a root domain instead of the `www` subdomain is purely a vanity choice of mine. Whether or not you should use one is a debate with plenty people in both[6] camps[7]. Picking one or the other, however, will at least be less confusing to bots and users.

The second `server` block defines the root (where the web files are located). The `index` files to look for first if no file is defined in the URL explicitly. We then set the used character sets to utf-8, followed by defining an access and error log to be used specifically for this site.

The `location` block uses `try_files` to tell Nginx to use the given URI as it is. Then it tries the URI as a directory. Finally it returns a 404 response if no file is found.

Then we set specific rules for `favicon.ico` and `robots.txt` files. These two are often requested without our knowledge by browsers and search engine bots. We don't usually care to fill up our logs with information about if/when they are accessed.

Finally, I include something new - `h5bp/basic.conf`. As noted briefly above, I often use H5BP's excellent configuration for Nginx. These optional include support for:

- Setting expiration headers
- Setting the X-UA-Compatibility header for IE rendering
- Protecting System Files
- Setting up a cache-busting scheme
- Using CORS headers
- Other minor additions, which you can read about within the h5bp nginx repository[8].

H5BP Config

I don't use all of H5BP's config. but I do usually grab their specific configuration for handling files and useful HTTP headers. Here's a quick few commands to get the `h5bp/basic.conf` and related files:

[6] http://www.yes-www.org/

[7] http://no-www.org/

[8] https://github.com/h5bp/server-configs-nginx

```
# Download the repository
wget -O h5bp.zip https://github.com/h5bp/server-configs-nginx/archive/master.zip

# Uncompress it
unzip h5bp.zip

# Grab the "h5bp" sub-directory
sudo mv server-configs-nginx-master/h5bp /etc/nginx/

# Remove the rest we don't use
rm -rf h5bp.zip server-configs-nginx-master
```

If you inspect `/etc/nginx`, you'll see the `h5bp` directory in there. We discarded the rest of the repository that we downloaded above.

Once that's included, the nginx config `include h5bp/basic.conf` will work. Here is what's included in `basic.conf`:

```
include h5bp/directive-only/x-ua-compatible.conf;
include h5bp/location/expires.conf;
include h5bp/location/cross-domain-fonts.conf;
include h5bp/location/protect-system-files.conf;
```

This sets the `X-UA-Compatible` header to `IE=Edge` to get the latest IE compatibility (document mode).

Then `expires.conf` sets some good defaults for the myriad of static files the server might serve.

The `cross-domain-fonts` configuration allows cross-domain requests. It then sets cache directive for the web fonts. If you're using hosted web fonts, I'd suggest removing this. It first sets the `Access-Control-Allow-Origin` header to `*`, which can open your site up to XSS attacks.

Finally the `protect-system-files` configuration disables access to dot files. This includes numerous files which might appear on the web server accidentally (.sql, .ini, .psd, .sh and many others).

You may also want to use h5bp's `mime.types` file instead of the stock Nginx file. This file is found at `/etc/nginx/mime.types`. The H5BP version has more file types defined.

Enabling the Site

Once your configuration is all set, you can save it and then add it to the `sites-enabled` directly via a symlink:

```
sudo ln -s /etc/nginx/sites-available/serversforhackers \
     /etc/nginx/sites-enabled/serversforhackers
```

Once the symlink is created, you can reload Nginx to make the configuration take affect:

```
sudo service nginx reload
```

That's it for a static site! We've setup the basics of handling a static HTML site and set some sane defaults for caching and basic protection. Read more on the H5BP repository if you want to customize these further for your needs.

Dynamic Site

The dynamic site I have is fideloper.com, which is a PHP application. In this section, we'll use what we learned in the Nginx chapter to proxy dynamic requests. In this case, we'll proxy requests off to the FastCGI process, PHP-FPM.

PHP-FPM

Let's install PHP-FPM, as per our previous chapter. We'll cover this quickly as the details are outlined in the Nginx chapter.

```
# add a repo to get the latest stable PHP
sudo add-apt-repository -y ppa:ondrej/php5
sudo apt-get update

# Install PHP-FPM and other PHP modules
sudo apt-get install -y php-fpm php-cli php5-mysql \
     php5-curl php5-gd php5-mcrypt php5-memcached
```

By default, PHP-FPM will be listening on a UNIX socket. I usually change this to a TCP socket. That configuration is found in `/etc/php5/fpm/pool.d/www.conf`. Let' use this find & replace one-liner to change that:

```
sudo sed -i "s/listen =.*/listen = 127.0.0.1:9000/" \
     /etc/php5/fpm/pool.d/www.conf
```

I'm going through this quickly - read the related chapters on PHP and Nginx for details on what's going on here. The focus of this case study is on Nginx configuration rather than PHP.

Once that's setup, we're ready to integrate Nginx with our PHP application (using PHP-FPM).

Virtual Host

The Nginx configuration is largely the same as the static site, except we'll define what happens when PHP files are called:

File: /etc/nginx/sites-available/fideloper

```
server {
    listen 80;

    server_name *.fideloper.com;
    return 301 $scheme://fideloper.com$request_uri;
}

server {
    listen 80;

    server_name fideloper.com;

    root /var/www/fideloper.com/public;
    index index.html index.htm index.php;

    charset utf-8;

    access_log /var/log/nginx/fideloper.com.log;
    error_log  /var/log/nginx/fideloper.com-error.log error;

    include h5bp/basic.conf;

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

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    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
```

```
37          fastcgi_param PATH_INFO       $fastcgi_path_info;
38          fastcgi_param ENV production;
39      }
40
41  }
```

Most of this is the same as the static site, but with a few tweaks.

We're using the fideloper.com domain rather than serversforhackers.com. We can use the same redirect rule and just change the server_name, log and document root directives as needed.

Another difference here is that I add `index.php` to the `index` directive. This tells Nginx to try the `index.php` file if only a directory is given in the URI.

In conjunction with this, I've added `/index.php$is_args$args` to the try_files directive. This will try the index.php file with any GET variables added. The `$is_args` variable adds a "?" to the URL if any url parameters are present. The `$args` variable contains any URL parameters included.

The `try_files` directive specifying `index.php` is the primary way to make "pretty urls". It eliminates the need of having `index.php` in our URLs.

Finally we have a location block which will take any requests for a PHP file and parse it using PHP-FPM. It passes to PHP-FPM via the TCP Socket we defined in its configuration above. The particulars of each of these directives is covered in the chapter on Nginx, however let's quickly cover some:

fastcgi_split_path_info splits the URI into two regex capture groups - the path and the script filename.

fastcgi_pass tells Nginx where to pass the request to. Here we're passing to Nginx which listens at `127.0.0.1:9000`.

include fastcgi.conf includes Nginx's out-of-the-box fastcgi parameter configuration. This sets lots of variables which PHP's uses in its `$_SERVER` global.

fastcgi_param lets us add our own parameters or override those found in the `fastcgi.conf` configuration file. Here we are setting `PATH_INFO`, which was parsed using `fastcgi_split_path_info`. Lastly I set an application environment variable. My application uses this to detect what environment to run the application in. Each of these parameters become environmental variables our applications can use.

Security

We're all used to allowing any PHP file on our server run. The `location ~ \.php$` block in this configuration does just that - any URI ending in `.php` will be parsed as PHP.

However this is a convenient place to lock that down. For example, if our application only has an `index.php` file, we can ensure *only* those files are parsed as PHP. This can help prevent server attacks if a rogue PHP file somehow finds its way onto our server.

An example of that can be found in the recommended setup for Symfony applications. This only allows `app.php`, `app_dev.php` and `config.php` as access points to the application:

```
# Allow app.php to be "hidden" like we do with index.php
location / {
    try_files $uri /app.php$is_args$args;
}

# Only parse app/app_dev/config.php files.
location ~ ^/(app|app_dev|config)\.php(/|$) {
    # Other parameters as normal
}
```

Enabling the Site

Once again, when your configuration is all set, you can save it and then add it to the `sites-enabled` directly via a symlink:

```
sudo ln -s /etc/nginx/sites-available/fideloper /etc/nginx/sites-enabled/fidelop\
er
```

Once the symlink is created, you can reload Nginx to make the configuration take affect:

```
sudo service nginx reload
```

Wrapping Up

That's really all there is to a basic Nginx setup! Nginx is really simple to configure. For more details, see the chapter on Nginx, which covers these in a bit more depth, along with other options and uses for Nginx.

We have setup two sites on the same server, and configured them separately.

These two configurations have served me very well for a long time, even though Nginx can do much more!

Case Study: MySQL Replication

In this scenario, we'll setup MySQL database replication. We'll have a Master database and a Replica database, which is a live copy of the Master database.

When MySQL is used this way, the Replica database reads the "binlog" of the Master database. The Replica database will playback every query (minus select queries). This makes the Replica database an exact copy of the Master database.

Replication is not instant, but it is fast. However, a very write-heavy application can cause the Replica to get behind by a matter of seconds, minutes and even (in more extreme cases) hours.

A good time to use MySQL replication is with a read-heavy application. In this case, both databases can be read from at any time.

Assuming your Replica database does not get too far behind, you'll have a reasonable assurance that the two databases will be in a consistent state at any time.

In such a setup, the application can read from either database, but needs to write **only** to the Master database.

Writing to the Replica database will break replication - the writes to the Replica will not be added to the Master database.

In this case study, we'll look into setting up MySQL Replication. We'll distribute the traffic between the read servers using the HAProxy load balancer. Finally, we'll secure the database servers with firewall rules.

The serversforhackers.com[9] video site covers the following as well, and will soon expand into similar topics for PostgreSQL!

MySQL Replication

Let's pretend we have three servers in this setup:

- The Master Database: 172.17.0.2
- The Replica Database: 172.17.0.3
- The Load Balancer: 172.17.0.4
- Database Name: "my_app"

[9] https://serversforhackers.com

Master Database

First, we'll setup and install the Master database on server `172.17.0.2`.

```
sudo apt-get update
sudo apt-get install -y mysql-server mysql-client
```

> In production, the root password should be made a strong password. We'll use other logins to access the database later. User "root" should only be used locally within a database server.

Next, we'll edit this database's /etc/mysql/my.cnf‘ config file to make it ready to be the Master:

File: /etc/mysql/my.cnf

```
# Bind to local network rather than just loopback
bind-address    = 172.17.0.2

# Give the server an ID (any ID is fine)
server-id       = 1

# Setup the binlog to be written to:
log_bin         = /var/log/mysql/mysql-bin.log
binlog_do_db    = my_app      # Choose the database to replicate
```

Here's what we did there:

`bind-address`: By default, MySQL binds to the loopback interface (`127.0.0.1`, the localhost network). This means it won't be able to listen on other networks for connections.

To change this, we can add a second `bind-address` line to also connect to remote connections. That will look like this:

File: /etc/mysql/my.cnf

```
bind-address    = 127.0.0.1
bind-address    = 172.17.0.2
```

`server-id`: We must set a unique server ID per database when using more than one database. The ID is arbitrary - many times you'll simply see an incrementing integer.

`log_bin`: This is the location of the binlog. The Relica database will read the binlog and play back all the write queries made to the database.

`binlog_do_db`: This defines which databases to be used for replication. Note that the use of this comes with some caveats[10]. MySQL sends all database information into the binlog, but all but this defined database gets filtered out. It doesn't get filtered out correctly in all cases. Read the previous link for more information.

Once the `/etc/mysql/my.cnf` file is edited and saved, we can restart MySQL to get the new settings in place:

Restart MySQL

```
1 sudo service mysql restart
```

Next, we'll create a user for the Replica database to use to connect to the Master database.

Grant replication rights

```
1 mysql -u root -p -e "GRANT REPLICATION SLAVE ON *.* TO 'replica_user'@'172.17.0.\
2 3' IDENTIFIED BY 'password';"
```

The username can be anything. Make sure to use a strong password - I just used "password" in this case.

Additionally, note that instead of the wildcard `%` hostname, the IP address of the Replica server was used. The Master database will only accept connections from user `replica_user` from server `172.17.0.3`.

Then flush privileges to ensure the updates take place:

```
1 mysql -u root -p -e "FLUSH PRIVILEGES;"
```

Now, still on the same server, log into MySQL client. We'll need to run a few commands within the same session, where as before we were making some one-shot queries:

```
1 mysql -u root -p
```

Once logged in, we'll create our database and get some statuses information. Note that we're creating a new database. However, the following instructions will work for existing databases as well.

[10] http://www.mysqlperformanceblog.com/2009/05/14/why-mysqls-binlog-do-db-option-is-dangerous/

```
mysql> CREATE DATABASE my_app;
mysql> USE my_app;
mysql> FLUSH TABLES WITH READ LOCK; -- Database won't be writable now!
mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000001 |      496 | my_app       |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

This creates a database, then uses it, then makes the database read-only. Finally, we get the status of the Master database. Keep track of the "File" (the binlog) and "Position". These mark where the Replica server will start reading from in the binlog.

Next we're going to export the data from the Master database. Open a **NEW** terminal/shell session on the same server. We need to do this while the READ LOCK (read-only) mode stays on. What we'll do is dump out the database in it's current state. This is important if you have a previously existing database. We need to get our replicate database up to speed before it continues reading from the Master database.

Once in your Master database server in a new terminal session, we'll use `mysqldump` to make a copy of our new database:

```
mysqldump -u root -p --opt my_app > my_app.sql
```

> Read about what –opt does here[11].

Then you can return to your first session/terminal windows and unlock the MySQL tables so the database can continue to be used. Since we know the binlog file and position to start from, we can let the Master database be written to now.

Within your previous session, which should still be logged into the MySQL client:

```
mysql> UNLOCK TABLES;
mysql> exit;
```

Next, we'll copy the my_app.sql file that we just created from the Master database server to the Replica server:

[11] http://dev.mysql.com/doc/refman/5.1/en/mysqldump.html#option_mysqldump_opt

```
# Your servers will need permission to
# "talk" to eachother over SSH
scp my_app.sql username@172.17.0.3:~/
```

Replica Database

Now, log into the Replica server. We'll import the database we just dumped and also setup this server to be used as a Relica.

Log into the MySQL server to create the database:

```
mysql -u root -p
> CREATE DATABASE my_app;
> exit;
```

Now we can import our database file we just copied over from the Master database:

```
mysql -u root -p my_app < ~/my_app.sql
```

Then we'll configure this database to work as a Replica:

File: /etc/mysql/my.cnf

```
# Bind to local network rather than just loopback
bind-address  = 172.17.0.3

# Anything but what the Master db is
server-id     = 2

# Setup the bin log
log_bin       = /var/log/mysql/mysql-bin.log
relay-log     = /var/log/mysql/mysql-relay-bin.log

# Define the database to replicate
binlog_do_db = my_app
```

Once again, let's go over what we're editing here:

`bind-address`: The Replica database doesn't need to listen on anything but the localhost loopback interface in order to work as a Replica. However, we'll have it listen to connections in order for HAProxy to connect to it when we use it for read queries:

File: /etc/mysql/my.cnf

```
bind-address    = 127.0.0.1
bind-address    = 172.17.0.2
```

`server-id`: Once again, we'll set a unique server ID

`log_bin`: Give the location of the binlog

`relay-log`: The relay log is a copy of the binlog that the Replica makes. It then reads from the relay-log to replay the events/queries to become in-sync with the Master database.

`binlog_do_db`: The Replica should also define which database to replay events from, if you're using the `binlog_do_db` option.

Save the `/etc/mysql/my.cnf` file and exit. Then restart MySQL:

```
sudo service mysql restart
```

Then we can tell the Replica database to start the reading from Master:

```
$ mysql -u root -p
> CHANGE MASTER TO MASTER_HOST='172.17.0.2',MASTER_USER='replica_user', MASTER_P\
ASSWORD='password', MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=496;
> START SLAVE;
> SHOW SLAVE STATUS;
```

Note that the MASTER_LOG_FILE and MASTER_LOG_POS options are set to the values we received from the Master database via the `SHOW MASTER STATUS;` query.

> MySQL explicitly calls this a "Master-Slave" setup, and so "SLAVE" is used in some commands above, as the Replica database is called a Slave

That should start replication!

If you're following along as an example and have an empty database, you can try the following in the Master database:

```
CREATE TABLE users (
        id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        user varchar(255) NOT NULL,
        password varchar(255) NOT NULL
) ENGINE=InnoDB;
```

That will create a simple users table. Check that it exists in both the Master and Replica database. Then add a new row:

```
INSERT INTO        users (user, password) VALUES ('fideloper', 'secret');
```

Once that's complete, check both the Master and Replica database to ensure the data is in both.

Firewalls

Next we want to ensure we have some proper rules setup. For our databases, we only need to allow SSH connections and TCP port 3306, MySQL's default port used. Since these servers likely aren't being used for other needs, we can close off all other connections.

What we're doing here exactly is covered in the book's chapters on setting up Firewalls.

On the Master database server, we'll allow the loopback interface to communicate. Then we'll allow established/existing and SSH connections.

Finally we'll open up TCP connections from the Replica server *and* the HAProxy load balancer. Both will be making connections to MySQL on port 3306. Finally, we'll drop any other traffic:

```
sudo iptables -A INPUT -i lo -j ACCEPT
sudo iptables -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
sudo iptables -A INPUT -p tcp -s 172.17.0.3 --dport 3306 -j ACCEPT # Replica Ser\
ver
sudo iptables -A INPUT -p tcp -s 172.17.0.4 --dport 3306 -j ACCEPT # HAProxy LB
sudo iptables -A INPUT -j DROP
```

On the Replica database server, we'll do similar. Incoming connections will only be made by the HAProxy load balancer. We'll only allow incoming connections from that server in addition to allowing SSH connections.

```
sudo iptables -A INPUT -i lo -j ACCEPT
sudo iptables -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
sudo iptables -A INPUT -p tcp -s 172.17.0.4 --dport 3306 -j ACCEPT
sudo iptables -A INPUT -j DROP
```

That should do it for Firewall rules. Be sure to save these rules using the `iptables-persistent` package as outlined in the Firewall chapter.

HAProxy

We'll balance READ traffic between the two databases using HAProxy.

In **both** Master and Replica database, add a HAProxy user which will simply connect to check if the database is available:

```
mysql -u root -p
mysql> USE mysql;
mysql> INSERT INTO user (Host,User) values ('172.17.0.4','haproxy_check');
mysql> FLUSH PRIVILEGES;
```

Then we can configure HAProxy to listen on both MySQL servers, and run health checks:

```
sudo vim /etc/haproxy/haproxy.cfg
```

Set a frontend for the databases:

File: /etc/haproxy/haproxy.cfg

```
frontend mysql
        bind *:3306
        mode tcp
        option tcplog
        default_backend databases
```

This is fairly simple. We'll listen on MySQL's default port of 3306. We'll use TCP mode over HTTP, since these are TCP connections, not HTTP. This necessitates the use of the tcplog over the httplog. Finally we'll tell it to use the "databases" backend, which we'll define next.

Then define a backend with our two database servers:

File: /etc/haproxy/haproxy.cfg

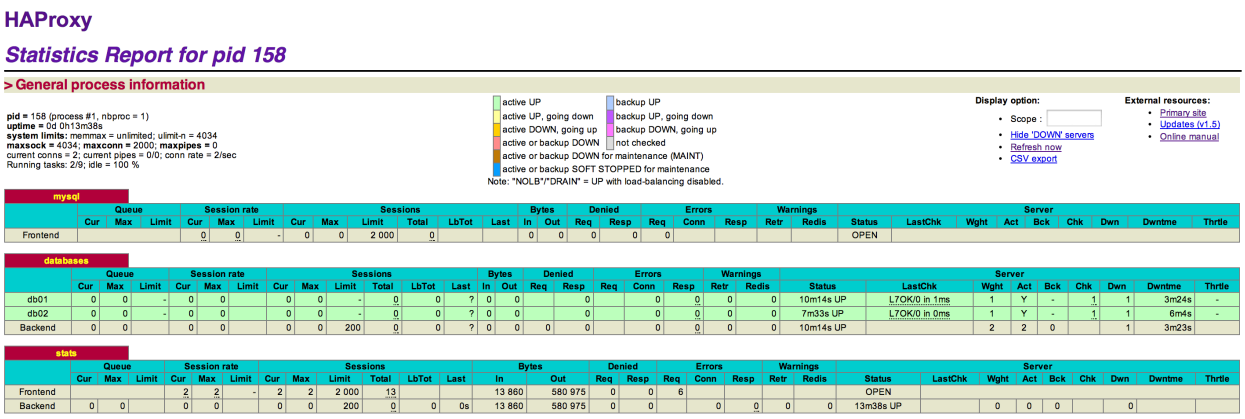
```
backend databases
        mode tcp
        option tcplog
        balance leastconn
        option mysql-check user haproxy_check
        server db01 172.17.0.2:3306 check
        server db02 172.17.0.3:3306 check
```

Once again, we use TCP mode and set it to log using the tcplog format. Then we set it to balance on the "leastconn" (least connections) balancing algorithm. This is a good algorithm to use for long-running connections, such as database connections.

Next, we'll use HAProxy's "mysql-check", with the username "hapoxy_check" we created. This tests if HAProxy can connect to the database, but knows nothing of the databases state. It won't tell us if the data in the database is corrupt.

Finally we define our two database servers, giving them a name ("db01", "db02"). We'll tell HAProxy to run the "mysql-check" health checks.

If you define the web interface for monitoring, you can check that HAProxy can speak to these databases:



HAProxy

Statistics Report for pid 158

> General process information

pid = 158 (process #1, nbproc = 1)
uptime = 0d 0h13m38s
system limits: memmax = unlimited; ulimit-n = 4034
maxsock = 4034; **maxconn** = 2000; **maxpipes** = 0
current conns = 2; current pipes = 0/0; conn rate = 2/sec
Running tasks: 2/9; 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

mysql

	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	0	-	0	0	2 000	0			0	0	0	0	0					OPEN							

databases

	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
db01	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	10m14s UP	L7OK/0 in 1ms	1	Y	-	1	1	3m24s	-
db02	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	7m33s UP	L7OK/0 in 0ms	1	Y	-	1	1	6m4s	-
Backend	0	0		0	0		0	0	200	0	0	?	0	0	0	0		0	0	0	0	10m14s UP		2	2	0		1	3m23s	

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				2	2	-	2	2	2 000	13			13 860	580 975	0	0	6					OPEN							
Backend	0	0		0	0		0	0	200	0	0	0s	13 860	580 975	0	0		0	0	0	0	13m38s UP		0	0	0		0		

> If HAProxy absolutely won't connect to the database servers, be sure to periodically log into the database servers and run `mysqladmin -u root -p flush-host` on each server. If there's too many bad connections (HAProxy checks every few seconds) from HAProxy, it will get locked out of the database server.

The Application

Any application in this setup should differentiate read vs write connections in app logic, so as not to write to the Replica database, ever.

Read connections point to the load balancer instead of directly to the database servers. Write connections can connect directly to the Master database. If you wanted to, you could define a another frontend and backend within HAProxy just for the Master server. The backend would just have one server in that case.

In any case, both of our databases need a user which can accept connections from HAProxy for reads. MySQL will see a load-balanced connection as coming from the load balancer. For the Master server, the user should also be allowed to connect from our application server(s).

If all of our servers are within the same class C subnet (all but the fourth section of a IP address are the same), then we can make this pretty easy.

On the Master server, we can grant all privileges to our application user:

```
$ mysql -u root -p
mysql> CREATE USER 'my_app_user'@'172.17.0.%' IDENTIFIED BY 'some_secure_passwor\
d';
mysql> GRANT ALL PRIVILEGES ON my_app.* TO 'my_app_user'@'172.17.0.%';
mysql> FLUSH PRIVILEGES;
```

Note that we are granting access to the user coming from any subnet of the network our servers are all on. In this example: `172.17.0.%`.

> You might want to be more restrictive on what privileges you grant[12].

This user will actually get replicated to the Replica server so we don't need to create a new user there.

Let's see what some faux-connections would look like in an application:

```
# Define read connection
readConnection = new DbConnection('172.17.0.4:3306', \
    'my_app_user', 'some_secure_password', 'my_app')

# Write Connection
writeConnection = new DbConnection('172.17.0.2:3306', \
    'my_app_user', 'some_secure_password', 'my_app')
```

Note that the write connection connects directly to the Master database. The read connection connects to HAProxy, which will then distribute the MySQL connections between the two available databases.

It's up to your application to handle managing read vs write connections appropriately.

[12] http://dev.mysql.com/doc/refman/5.1/en/privileges-provided.html#priv_usage

Case Study: Python & uWSGI

Python is a highly-used language in the web. Django, Flask and other Python-based applications are often hosted with Nginx or Apache.

Python's PEP[13] (Python Enhancement Proposals) 3333 defines the WSGI standard. This is a standard interface that can be used to serve Python on the web.

This is similar to CGI, FastCGI or other gateway interfaces. These sit between a web application and your web server and are able to convert web requests into requests on your application code.

While PHP-FPM implements FastCGI for PHP, uWSGI[14] implements WSGI for Python.

uWSGI is a versatile tool. In addition to WSGI, it can speak FastCGI and HTTP as well!

In this case study, we'll see how to host an application written in Python's Flask. We'll use uWSGI to pass requests to a sample application from Apache and Nginx.

The Application

We'll write a quick Flask application for demonstrate. You'll likely have a more complex application to host. Luckily an application's complexity won't affect how we use our web servers.

If you're interested in this Case Study, I'm going to assume you're at least familiar with Python and some of its common tools. Specifically we'll be using PIP and Virtualenv.

To write our application, we need to install some Python dependencies and then code a small web application.

I'll be using Ubuntu 14.04. First, install Virtualenv and PIP:

Installing Virtualenv and PIP

```
1 sudo apt-get install -y python-virtualenv python-pip python-dev
```

Then we can use Virtualenv to setup a Python environment named "flaskapp":

[13] http://legacy.python.org/dev/peps/

[14] https://uwsgi-docs.readthedocs.org/en/latest/

Create virtualenv 'flaskapp'

```
mkdir ~/python
cd ~/python
virtualenv flaskapp
cd ~/python/flaskapp
source bin/activate
```

This creates and enables a Python virtual environment. With a virtual environment, we can use any Python version and library of our choosing. We can setup multiple environments if we have several applications with conflicting dependencies.

In this example, we'll use the default Python version, 2.7.

Next we can use PIP to install Flask:

Install Flask into environment 'flaskapp'

```
cd ~/python/flaskapp
bin/pip install flask
```

Finally we can create a small Flask application. Create the directory `~/python/flaskapp/testapp`:

Creating ~/python/flaskapp/testapp

```
mkdir ~/python/flaskapp/testapp
cd ~/python/flaskapp/testapp
```

Then we can code our application in file `__init__.py`:

File: ~/python/flaskapp/testapp/init.py

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

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

Finally, save the application and test it out:

Running the example Flask application

```
$ python __init__.py
* Running on http://0.0.0.0:5000/
```

You can run the command `curl localhost:5000` in your server or check the site in your browser to see the "Hello World!" message.

Once you see that working, you can cancel out of it using <Ctrl+c>. Running the application in that manner is only for testing. We'll see how we might run this in production next!

uWSGI

Next we need to install the Gateway Interface for our web server to 'speak to'. This will translate a web request from a web server to our Python application.

We should install uWSGI in the same environment as our Flask application. If you're no longer in the environment setup above, re-source it by going to the "flaskapp" environment and activating it:

Re-activating a virtual environment

```
cd ~/python/flaskapp
source bin/activate
```

Then install uWSGI using PIP:

```
cd ~/python/flaskapp
bin/pip install uwsgi
```

Once that's installed, we can test out uWSGI by having it listen for HTTP connections. You'll note that this is similar to us just running the the application directly as we did above.

Using uWSGI has advantages, however. For example, uWSGI can create multiple instances of the application. It will create multiple processes and balance requests amongst them!

Here's a quick example, run from the `~/python/flaskapp` directory:

Using uWSGI to run the Flask application

```
cd ~/python/flaskapp
uwsgi --http :9000 --module testapp --callable app
```

This command does the following:

- Listens for http requests on any network interface on port 9000
- Looks for the Python module `testapp`
- Looks for a callable named `app`, which Flask sets up for us (this is a default setting of Flask)

Now if we run the command `curl localhost:9000` or go to our server at port 9000 in our browser, we should see the "Hello World" page again!

This is great to start, but let's use uWSGI another way as well. In a production environment, we'll use the WSGI protocol.

Using HTTP over WSGI is just as viable an option. The WSGI protocol may be more convenient or performant for many Python applications.

The command to use WSGI can look like this:

Using the WSGI protocol to speak to the Flask application

```
1 cd ~/python/flaskapp
2 uwsgi --socket 127.0.0.1:9000 --module testapp --callable app \
3       --stats 127.0.0.1:9191 --master  --processes 4 --threads 2
```

This command will do the following:

- Create a TCP socket at `127.0.0.1:9000`. We specify localhost so external networks cannot listen on this directly.
- Once again use the Python module `testapp`.
- Once again use the "callable" of `app`.
- Show stats on `127.0.0.1:9191`, which will output a JSON object of uWSGI statistics. An API client or server monitoring software can consume this API.
- Set uWSGI as a master process that should create 4 processes, 2 threads per process.

We use the `--master` flag to tell uWSGI to act as a master process. It will managing the processes and threads. Note that a setup with multiple threads assumes your application is thread-safe.

Note that we can't use `curl` or our browser to test this. It's listening on a TCP socket, but it's expecting a WSGI request rather than an HTTP request.

We can put this command into a ini-style configuration as well. This might be more useful to use as a template. You can include this in any configuration management software. We'll use a configuration file and specify settings more appropriate for production use.

Create file `~/python/flaskapp/uwsgi.ini`:

uWSGI configuration file ~/python/flaskapp/uwsgi.ini

```
[uwsgi]
socket = 127.0.0.1:9000
module = testapp
callable = app
threads = 4
processes = 2
master = true
stats = 127.0.0.1:9191
chdir = /home/vagrant/python/flaskapp
virtualenv = /home/vagrant/python/flaskapp
uid = www-data
gid = www-data
```

We essentially have the same setting as above, except we give uWSGI some extra parameters. First, tell uWSGI to change into the directory `/home/vagrant/python/flaskapp`. We'll instruct it to use the virtualenv found in the same directory. This way we can ensure the application is running with the correct version of Python and libraries.

Finally, we have uWSGI run its processes as user and group `www-data`. We don't want our applications to run as a privileged user for security reasons.

You'll need to adjust your file paths to suit your needs. In this example, I tested on a Vagrant server, and thus was logged in as user Vagrant.

We can start uWSGI with the following command:

```
uwsgi --ini /home/vagrant/python/flaskapp/uwsgi.ini
```

Monitoring uWSGI

Now we know how to run our Python application using uWSGI and the WSGI protocol.

The next step will be to use Apache or Nginx to proxy web requests to this application. Before we do that, we need to ensure uWSGI can start on system boot and restart it when needed.

There are many options for monitoring processes. We could use Upstart, SysV, Systemd, Circus, Supervsord, and Monit just to name a few. I find that the simplest way to monitor uWSGI is with Supervisord.

First we can install Supervisord:

```
sudo apt-get install -y supervisor
```

For more information on the specifics of Supervisord, see the Process Monitoring chapter.

Then we can create a configuration for uWSGI inside of `/etc/supervisor/conf.d/`. Let's create a file `uwsgi.conf`:

Supervisord configuration for uWSGI at /etc/supervisor/conf.d/uwsgi.conf

```
[program:uwsgi]
command=/home/vagrant/python/flaskapp/bin/uwsgi \
        --ini /home/vagrant/python/flaskapp/uwsgi.ini
directory=/home/vagrant/python/flaskapp
autostart=true
autorestart=true
startretries=3
stderr_logfile=/var/log/flask/testapp.err.log
stdout_logfile=/var/log/flask/testapp.out.log
user=www-data
```

This will run the uWSI command. We set uWSGI to start when Supervisord starts (on system boot), and to auto-restart if the process fails. It will give it 3 restart attempts before putting uWSGI into a "failed" state.

Note that we are running uWSGI as user `www-data` as well.

We need to create the log file directories, as Supervisord will not create them for us:

```
sudo mkdir /var/log/flask
```

Then we can have Supervisord start the uWSGI process and check its status:

Enable and start Supervisord-monitored uWSGI

```
$ sudo supervisorctl reread
$ sudo supervisorctl update
$ sudo supervisorctl start uwsgi
$ sudo supervisorctl status
uwsgi                            RUNNING    pid 5681, uptime 0:00:39
```

Great, now our Flask application is running! uWSGI is listening on localhost port 9000. Let's run through using Apache and Nginx to send requests to this application!

Apache

The Apache chapter explains the details of this. We'll go through it quickly here.

We'll cover:

- Installing Apache
- Enabling the appropriate Apache modules
- Creating a virtualhost to pass requests to uWSGI

Start by installing Apache and its uWSGI module:

Installing Apache and the Proxy-uWSGI module

```
sudo add-apt-repository -y ppa:ondrej/apache2
sudo apt-get update
sudo apt-get install -y apache2 libapache2-mod-proxy-uwsgi
```

Then enable the needed Apache modules:

Enable modules Proxy and Proxy-uWSGI

```
sudo a2enmod proxy proxy_uwsgi
sudo service apache2 restart
```

Finally, create a Virtualhost for the site. We'll create this at `/etc/apache2/sites-available/002-flaskapp.conf` for this demonstration:

Virtualhost file: /etc/apache2/sites-available/002-flaskapp.conf

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com

    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks +MultiViews
        AllowOverride All
        Require all granted

        <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>
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/examle.com-error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/examle.com-access.log combined

</VirtualHost>
```

This virtualhost is listening for requests to `example.com`. You should of course change the domain and file paths as needed for your own usage.

The following `Location` directive takes any request for the `/static` directory, or file within, and attempts to serve it as a static file. It will not send the request to the application:

Informing Apache to serve files out of the `/static` directory as if they are static files

```
<Location /static>
    ProxyPass !
</Location>
```

The remaining `Location` and `Proxy` blocks let you serve all other request to the Python application:

Proxying requests to uWSGI

```
<Proxy *>
    Require all granted
</Proxy>
<Location />
    ProxyPass uwsgi://127.0.0.1:9000/
    ProxyPassReverse uwsgi://127.0.0.1:9000/
</Location>
```

Once this is created, symlink the Virtualhost to the `sites-enabled` directory.

```
sudo a2ensite 002-flaskapp

# The above command is equivalent to:
sudo ln -s /etc/apache2/sites-available/002-flaskapp.conf \
           /etc/apache2/sites-enabled/002-flaskapp.conf
```

Reload Apache's configuration to read in the enabled site. We can then run the `curl localhost` command or view the site in a browser:

Reloading Apache configuration and testing web requests at port 80

```
sudo service apache2 reload
curl localhost
```

I'm not showing DNS setup to point the domain to your web server here, but that is likely a step you need to take for production use.

Nginx

Nginx configuration is covered in detail in the Nginx chapter. We'll quickly cover the Nginx configuration needed to host a Python application.

Begin by installing Nginx:

Install Nginx on Ubuntu 14.04

```
sudo add-apt-repository -y ppa:nginx/stable
sudo apt-get update
sudo apt-get install -y nginx
```

Then we can create an Nginx virtualhost for the site:

File: /etc/nginx/sites-available/flaskapp

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root /var/www/example.com/public;
    index index.html index.htm;

    server_name example.com www.example.com;

    charset utf-8;

    location / {
        try_files $uri $uri/ @proxy;
    }

    location @proxy  {
        include uwsgi_params;
        uwsgi_pass 127.0.0.1:9000;
        uwsgi_param ENV production;
    }
}
```

This listens on port 80 for both IPv4 and IPv6 connections. We've set this virtual host as the default server. If the server receives a request without a correct `Host` header or directly from it's IP address, this site will be served.

The first `Location` block defines how to treat all incoming requests.

The `try_files` directive first attempts to fulfill the URI as is. It looks for a static file matching the URI, relative to the configured `root` directory.

If it does not find a file, it attempts to run the URI as a directory. The filenames found in the `index` directive are used to try to match files if the URI is a directory.

Failing to match the URI to an existing file or directory, it reaches the `@proxy` alias, which is defined below it. Nginx passes the request to the `location @proxy` directive. This, in turn, passes the request to uWSGI listening at localhost port 9000.

The use of `@proxy` is a more refined way to handle static files. Unlike Apache's configuration, we don't need to specify a directory used for static files. Instead, Nginx will pass any request that is not an existing file or directory to our application to handle!

Once this is virtualhost is saved, enable the site and reload Nginx's configuration:

```
# Add the site configuration to `sites-enabled`
sudo ln -s /etc/nginx/sites-available/flaskapp \
           /etc/nginx/sites-enabled/flaskapp

# Check for configuration errors
sudo service nginx configtest

# Reload Nginx configuration
sudo service nginx reload
```

Then you can test the site using `curl` or in a browser!

Wrap Up

That's about it for hosting a Python application.

Python's tooling has advanced enough to make hosting a Python application fairly trivial!

Most Python frameworks come with the ability to listen for WSGI requests. We usually just need to install a gateway capable of accepting requests and passing them to our application.

The Apache and Nginx web servers have made proxying requests to applications fairly simple!

For more specifics and explanation, read the Web Servers section of the Servers for Hackers eBook.

Case Study: Queues & Workers

Queues are a great way to take some task out of the user-flow and put them in the background. Allowing a user to skip waiting for these tasks makes our applications appear faster and more responsive.

For example, sending emails, deleting accounts or processing images are potentially long-running and memory-intensive tasks. They make great candidates for work which we can off-load to a queue.

PHP has a lot of libraries for using Queues. Laravel in particular makes them quite easy with its Queue package[15].

In this case study, we'll use the Beanstalkd[16] worker queue with Laravel to do image processing. We'll go over the steps to make this process production-ready.

This case studies covers:

1. Laravel and Queues
2. Installing Beanstalkd[17]
3. Churning through the queue of jobs with Laravel and Supervisor[18]

Laravel and Queues

From a code point of view, Laravel makes using queues very easy. Our application, the "producer", can simply run something like `Queue::push('SendEmail', array('message' => $message));` too add a "job" to the queue.

On the other end of the queue is the code and listener waiting for new jobs and a script to process the job. These are, collectively, the "workers". This means that in addition to adding jobs to the queue, we need to set up a worker to pull from the stack of available jobs. Laravel makes this easy as well.

we'll concentrate on the server concerns "in-between" the code. This includes setting up two servers to securely communicate to each other:

1. The application server will host the user-facing application. This application accepts web requests and adds jobs to the queue as needed; It's a producer.
    - This server will be at 192.168.99.2 in this example

[15] https://github.com/laravel/framework/tree/master/src/Illuminate/Queue
[16] http://kr.github.io/beanstalkd/
[17] http://kr.github.io/beanstalkd/
[18] http://supervisord.org/

2. The queue server will listen for new jobs and process them; It's a worker.
    - This server will be at 192.168.99.3 in this example
3. We could have a third server just for the Beanstalkd queue server. In this example, we'll simply install Beanstalkd on the worker server.

Beanstalkd is a "Broker". It listens for new jobs, storing them in a queue. Beanstalkd will then broker a job to the next workers to request one. The workers continuously poll Beanstalkd for new jobs once they've completed the last. You can use multiple workers in order to churn through a queue more quickly.

Application Code

The following applies to our **application server**, found at 192.168.99.2.

There are two pieces of code which concern us:

1. The application code adding new jobs (producers)
2. The application code consuming jobs (workers)

There are a few strategies to think about with our code.

We could create two separate code bases. This leaves us with two smaller projects.

Alternatively, we could write one code base and use it for both producing and consuming job.

For the sake of this demonstration, I'll assume we use one application. This is a common approach in Laravel. We simply load the whole application on both servers. Laravel has an optimization for Queues so this is not needlessly taxing on the queue server.

Laravel requires the `pda/pheanstalk` PHP package to use Beanstalkd. If you're a Laravel user, you'll need to run the following in your project:

```
composer require pda/pheanstalk ~2.0
```

We'll also be using the `imagine/imagine` package for image manipulation:

```
composer require imagine/imagine ~0.5.0.
```

A Consumer

Let's say our code has users who upload an image to their profile. This image likely needs resizing and cropping to fit into the site's layout. This can be a costly and slow operation. We don't want to make our users wait for it.

Instead, we'll upload the image to a central file store and add a new job to the queue. This job will resize the image. The user can be shown a temporary image or a loading graphic until the queue finishes processing the image.

Here's what that code might look like:

A user updates their profile, uploading an image

```
use Illuminate\Contracts\Filesystem;

class SomeController extends BaseController {

        protected $filesystem;

    public function __construct(Filesystem $filesystem) { ... }

    public function udateProfile()
    {
        // Profile updating code omitted...

        // Upload image to our central store
        $userImage = Input::file('user_image');
        $hash = md5( $userImage->getClientOriginalName().time() );
        $ext = $userImage->getClientOriginalExtension();
        $fileName = $hash.'.'.$ext;

        $this->filesystem->put(
            $fileName,
            file_get_contents($userImage->getRealPath())
        );

        // Produce new job for queue
        Queue::push(
            'App\Service\PhotoService',
            ['userid'   => $userId,
            'filename' => $fileName,
            'hash'     => $hash,
            'ext'      => $ext]
        );
```

```
32      }
33  }
```

This takes the uploaded image, re-names it to an md5 hash, and uploads it to a central file store. I usually use S3, but it just needs to be something both producers and workers can reach. We pass the minimal information needed into the queue, assuming the worker can use that to complete its task.

It's good practice not to send a lot of information (such as the whole image) into Beanstalkd and your workers. This needlessly uses a lot of memory.

Most libraries encode job data. Laravel happens to use data over JSON. This makes is possible for workers to be coded in any language.

A Worker

On the other end of this, we need to code a worker to process the images added to the queue.

The producer specified a `App\Service\PhotoService` class, which we'll create here. If no class method is specified, Laravel assumes the class will have a `fire()` method.

```
1   namespace App\Service;
2
3   use DB;
4   use Imagine\Image\Box;
5   use Imagine\Gd\Imagine;
6   use Illuminate\Contracts\Filesystem;
7
8   class PhotoService {
9
10      protected $filesystem;
11
12      public function __construct(Filesystem $filesystem) { ... }
13
14      public function fire($job, $data)
15      {
16          $rawimage = $this->filesystem->get($data['fileName']);
17
18          $image = new Imagine();
19          $image = $image->load( $rawimage->read() );
20
21          $resize = new Box(100, 100);
```

```
22          $thumb = $image->thumbnail($resize);
23
24          $resizedFileName = $data['hash'].'_100x100.'.$data['ext'];
25
26          $this->filesystem->put(
27              $resizedFileName,
28              $thumb->get($data['ext'])
29          );
30
31          DB::table('users')
32              ->where('id', $data['userid'])
33              ->update( ['image' => $resizedFileName] );
34
35          // Assuming success
36          $job->delete();
37      }
38
39  }
```

This is a basic worker. It will get the image data, pull down the image and process it. Finally it marks the job as complete. If the job is not marked as complete, it may be given back to the queue and re-processed!

You should consider adding code to inform the queue of a jobs status. If the job fails, you can tell Beanstalkd to retry the job later. If it's successful, you can inform Beanstalkd to delete the job. Don't forget to use error handling to catch errors and get the chance to put the job back into the queue.

Beanstalkd & Worker Server

On our application server we're adding a job to the queue when a user uploads an image. A worker will later process it.

In Laravel, we'll create a job by telling the Queue library what code will handle the job. In this case, that's the `fire()` method inside of the `App\Service\PhotoService` class.

Job added to the Queue from the Application Server

```
Queue::push(
    'App\Service\PhotoService',
    ['userId'   => $userId,
    'fileName' => $fileName,
    'hash'     => $hash,
    'ext'      => $ext]
);
```

This gets the job into the queue. Next we need to setup the queue and the workers to listen and respond to new jobs.

The following sections apply to our **worker server**, found at 192.168.99.3.

Install Beanstalkd

We'll start by installing Beanstalkd on our worker server.

Don't forget that I'm installing Beanstalkd on the same server as the workers for simplicity. In a distributed environment, I would put each piece on a separate server.

There's a PPA from Beanstalkd that has the latest stable version:

Installing Beanstalkd latest stable

```
sudo add-apt-repository ppa:beanstalkd/stable
sudo apt-get update
sudo apt-get install -y beanstalkd
```

Once that's installed, we're just about all set with it. We can test if it's running:

Check running processes for Beanstalkd

```
$ ps aux | grep beanstalkd
beansta+  1251  0.0  0.0  4472 616 ? S Sep16 0:00 \
        /usr/bin/beanstalkd -l 127.0.0.1 -p 11300
```

We can see that it's running, listening on localhost port 11300. This is a problem, as Beanstalkd will only listen for local connections. We're want Beanstalkd to be listening for communication from our application server!

Let's set Beanstalkd to listen on a network both servers can access. To do so, edit the default configurations setup for Beanstalkd, which is part of the APT package's SysV setup:

Original File: /etc/default/beanstalkd

```
BEANSTALKD_LISTEN_ADDR=127.0.0.1
BEANSTALKD_LISTEN_PORT=11300
```

Change the IP address to 0.0.0.0. This will set Beanstalkd listen on all networks.

Updated File: /etc/default/beanstalkd

```
BEANSTALKD_LISTEN_ADDR=0.0.0.0
BEANSTALKD_LISTEN_PORT=11300
```

Once that's saved, restart Beanstalkd and test to see if it's listening on all networks:

Check running processes for Beanstalkd after updating listen address

```
$ sudo service beanstalkd restart
$ ps aux | grep beanstalkd
... 00:39   0:00 /usr/bin/beanstalkd -l 0.0.0.0 -p 11300
```

Beanstalkd is now running and ready to accept new jobs! We'll cover securing this with the firewall in a bit, but first we can setup up the worker.

Process the jobs

At this point, we have an application with code for both producing the jobs and processing the job. It acts as both producer and worker.

Assuming our application is on the application server and running, our next step is to deploy the code into our worker server! We don't need a web server on the worker server. We can just install PHP and put the code on that server, which I'll leave up to you to do.

Once the application code is installed and deployed onto the worker server, we can have the application to listen for new jobs.

Laravel has it's own Queue Listening code, in the form of a PHP command-line call:

Available Artisan commands for queues

```
# CD into your deployed application:
cd /srv/app

# Listen for new jobs in the queue
# and fire them off one at a time
# as they are created
php artisan queue:listen
```

We run the `queue:listen` command to have Laravel listen to the queue and pull jobs as they become available. This is a "long-running" process, so we'll need a way to have this run continuously in the background.

By default, Laravel will run queue jobs synchronously. It runs the job at the time of creation.

This means the image will be processed in the same request that the user created when uploading an image. That's useful for testing, but not for production. We're making this process asynchronous by sending jobs to Beanstalkd on the worker server.

Now we can setup Laravel. In your `app/config/queue.php` file on the **application server**, set the default queue to 'beanstalkd':

File: app/config/queue.php

```
'default' => 'beanstalkd',
```

Then edit any connection information you need to change. I used the IP address of the worker server for the `host`. This lets Laravel on the application server know where to connect to Beanstalkd.

File: app/config/queue.php

```
'connections' => array(

        'beanstalkd' => array(
                'driver' => 'beanstalkd',
                'host'   => '192.168.99.3', # IP of the worker server
                'queue'  => 'default',
                'ttr'    => 60,
        ),

),
```

Now when we push a job to the queue from our application server, we'll be pushing to Beanstalkd!

You'll may also need to push this configuration update to your **worker server** as well. I suggest using environmental variables for configurations. That way specific configurations can be set per server rather than hard-coded in code.

Listening for Jobs

We've done the following on the **worker server**:

- Installed Beanstalkd
- Set it up to listen for remote connections
- Deployed our application code containing the worker code
- Updated the configuration on all servers to listen to the correct Beanstalkd server

Let's next start listening for jobs on the **worker server**:

Listening for new jobs in daemon mode

```
1 cd /srv/app
2 php artisan queue:work --daemon --sleep=3 --tries=3
```

I'm using `queue:work` in daemon mode, which will load in the application once. The queue workers won't need to reload the Laravel application for each new job.

This command tells the workers to sleep for 3 seconds between jobs. It will retry a job 3 times before "burying" it.

This is great, but this won't run in the background. We need a process monitor to watch this queue listener so that we don't have to keep it running ourselves.

Supervisord

Queues don't typically "push" jobs to their workers. Instead, a listener polls the queue for new jobs.

This is what the `$ php artisan queue:listen` and `$ php artisan queue:work` commands do. They poll for jobs and runs them as they become available.

As seen, however, these commands run in our console forever. If we log out, they stop! We need a way to make the server run the workers itself. This means we want the queue listener to start when the server starts and restart itself if it fails on an unexpected error.

This is where Supervisord comes in. It will ensure the workers are always running!

You can read more details on Process Monitoring and Supervisord in the Process Monitoring chapter.

Installing Supervisord

First, install Supervisord:

```
# Debian / Ubuntu:
sudo apt-get install supervisor
```

Next, configure it. We need to define a process to listen to. Supervisord configurations in Debian/Ubuntu typically are set in `/etc/supervisord/conf.d`. We'll create one called `queue.default.conf`:

File: /etc/supervisor/conf.d/queue.default.conf

```
[program:defaultqueue]
command=/usr/bin/php artisan queue:work --daemon \
        --env=your_environment
directory=/srv/app
autostart=true
autorestart=true
startretries=3
stderr_logfile=/var/log/queue/default.err.log
stdout_logfile=/var/log/queue/default.out.log
user=www-data
```

We now have a process called "defaultqueue" which we can tell Supervisord to start and monitor.

This Supervisord configuration set the following:

- `command` - The command to run.
- `directory` - The directory to `cd` into before running the command. This is useful if the worker code makes assumptions about file paths.
- `autostart` - Start the listener on system boot (supervisord boot).
- `autorestart` - Restart job if it fails.
- `startretries` - Only retry to start theprocess 3 times before considering it failed.
- `stderr_logfile` - Set the log file to send error output. We need to create the directories first, Supervisord won't create them for you.
- `stdout_logfile` - Where to send regular output - Again, create the log file directory if it doesn't exist. Supervisord will only create the file itself.
- `user` - Set this to run as user `www-data` to help prevent privilege escalation bugs.

Let's read in this configuration and start the listener:

```
sudo supervisorctl reread
sudo supervisorctl add defaultqueue
```

Now the "defaultqueue" process is on and being monitored! If the queue listener fails, Supervisord will restart the `php artisan queue:listen --env=your_environment` process.

You can check that it is indeed running that process with this command:

```
$ sudo supervisorctl status
defaultqueue      RUNNING    pid 3959, uptime 0:00:01
```

Or by checking processes:

```
$ ps aux | grep php

# You should see some output like this:
... 01:43   0:00 /usr/bin/php artisan queue:work --daemon --sleep=3 --tries=3
```

All set! Supervisord is monitoring the listener, which is polling Beanstalkd for jobs. New jobs are added when users edit their profile and upload an image.

Security

We left a potential security hole open. Beanstalkd on the worker server is listening on all networks! The firewall should be setup on the worker server to allow in as few connections as possible.

In this case, we only want our **application server** to be able to communicate with Beanstalkd on the **worker server**. We can use the iptables firewall for this.

The following will setup some firewall rules and enable their persistence through server reboots.

Iptables Rules

On the **worker server**, we'll create some rules to lock down all access except for SSH and Beanstalkd.

Each application server will need access to communicate with Beanstalkd on the worker server. We'll only allow access to Beanstalkd from our one Application server at address 172.17.0.3.

Run the following set of commands to setup basic firewall rules. These will allow SSH connections and access from the application server to Beanstalkd's port 3306.

Setting up iptables rules

```
sudo iptables -A INPUT -i lo -j ACCEPT
sudo iptables -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
sudo iptables -A INPUT -p tcp -s 172.17.0.3 --dport 3306 -j ACCEPT
sudo iptables -A INPUT -j DROP
```

Run `sudo iptables -L -v` to see a result similar to this:

Allowing traffic from SSH and the application server only

```
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
target  prot opt in     out  source        destination
ACCEPT  all  --  lo     any  anywhere      anywhere
ACCEPT  all  --  any    any  anywhere      anywhere ctstate RELATED,ESTABLISHED
ACCEPT  tcp  --  any    any  anywhere      anywhere   tcp dpt:ssh
ACCEPT  tcp  --  any    any  192.168.99.2  anywhere tcp dpt:11300
DROP    all  --  any    any  anywhere      anywhere

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


Chain OUTPUT (policy ACCEPT 24 packets, 1787 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

More information on Iptables and Firewalls can be found in the Security section of the eBook.

Persistence

Next we need to install the `iptables-persistent` package. Iptables rules are in-memory (RAM) only. This package allows Iptables rules to survive through a server reboot:

Install iptables-persistent

```
sudo apt-get install -y iptables-persistent
```

This should ask you if you want to save current rules. Select yes if so. Otherwise you can run the following:

Save iptables rules

```
sudo service iptables-persistent save
sudo service iptables-persistent start
```

This will save the rules to `/etc/iptables/rules.v4`. They will be read on system boot so the system firewall rules are reinitiated!

Log Management

We don't have any log management setup for the logs we're generating. You may have a log aggregator you use, in which case it would be wise to hook that up to read these generated log files.

However if you simply want to rotate your log files periodically, and delete old ones, you can use `logrotate` to do so for you. A simple setup would be like this:

File: /etc/logrotate.d/queue - Rotate all queue log files

```
/var/log/queue/*.log {
    daily
    missingok
    rotate 90
    compress
    delaycompress
    notifempty
    create 660 root root
    dateext
    dateformat -%Y-%m-%d-%s
}
```

This does the following:

- Applies to all files ending in `.log` in the `/var/log/queue` directory
- Runs daily
- If there are no log files, it won't generate an error
- Keep the last 90 days of logs on the server
- Compress old log files
- Delay compression of the rotated-out log file until 2nd time around rotating
- Don't rotate the logs if the log file is empty
- Create new log files as user/group root, with permissions set to `0600`

- Use a date extension for rotated log files, with the set format (`Y-m-d-s`)

And that's it! That will rotate the log file daily, compressing old ones and deleting any older than 90 days.