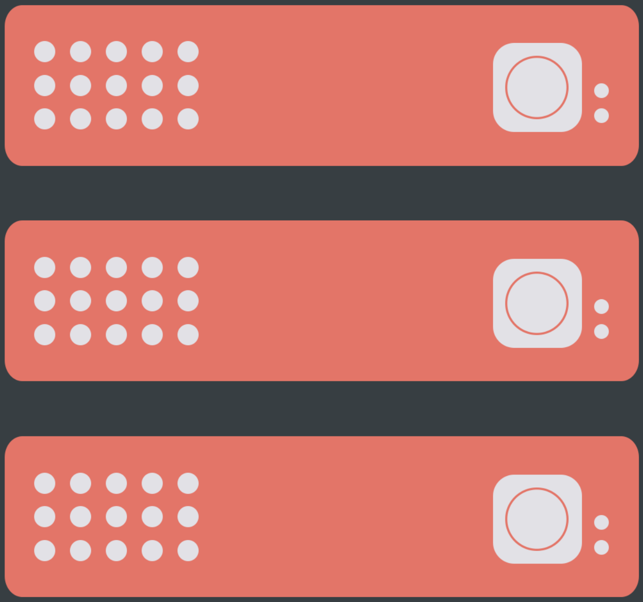


Servers for Hackers

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 2018-06-19

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:

#srvrsforhackers

Contents

Preview

The following chapters are selections of text you'll find in the Servers for Hackers eBook. These are just snippets of larger chapters, but I hope they show you some of the useful things you'll learn!

Visudo

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 can accomplish this### 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

```
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

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

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

Editing /etc/sudoers via visudo

```
%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

```
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 without passwords in production. It makes your privileged user as dangerous as giving root access.

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”!

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 bit.

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 by setting 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

Apache with FastCGI

Before Apache 2.4, we had to use `mod_fcgi` to send requests to a FastCGI gateway such as PHP-FPM. You still can actually, but it's not my preferred way.

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 `mod_proxy_fcgi`, which comes "out of the box" and is much simpler!

In this section, we'll look at using `mod_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 `mod_proxy_fcgi` to send PHP requests to the FastCGI gateway PHP-FPM.

First, we need to ensure the module is enabled:

```
# Let's disable mod PHP first:
sudo a2dismod php

# Then ensure mod_profyx_fcgi is enabled:
sudo a2enmod proxy_fcgi

# Install PHP-FPM:
sudo apt install -y php-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:

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.com/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.com/public/$1
```

The `mod_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/php/7.2/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 = /run/php/php7.2-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 = /run/php/php7.2-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/php/7.2/fpm/pool.d/www.conf
```

Lastly, as usual with any configuration change, we need to restart PHP-FPM:

```
sudo service php7.2-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 and overall more solid configuration.

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 php7

# Then ensure mod_profyx_fcgi is enabled:
sudo a2enmod proxy_fcgi

# Install PHP-FPM if necessary:
sudo apt-get install -y php7.2-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>

    <FilesMatch \.php$>
        SetHandler "proxy:fcgi://127.0.0.1:9000"
    </FilesMatch>

    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 new directive here is:

```
<FilesMatch \.php$>
    SetHandler "proxy:fcgi://127.0.0.1:9000"
</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:

```
<FilesMatch \.php$>
    SetHandler "proxy:unix:/run/php/php7.2-fpm.sock|fcgi://localhost/"
</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:/run/php/php7.2-fpm.sock|fcgi://localhost/"
    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 in order 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>
```

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

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 tell Apache to also enable the `proxy` module if it's not already enabled.

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

    <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>

    <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>
```

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 our 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.

You can adjust this directory path as needed, but it does have ramifications to your applications in terms of how static assets will be treated.

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.

The `Location` block "style" of proxying requests is best used for applications built in Python, Ruby or others which commonly use FastCGI gateways but don't have a file-based point of entry like PHP applications do.

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

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 `php7.2-fpm` for PHP 7 (there is not currently a `php-fpm` package):

```
sudo apt install -y php-fpm
```

As we mentioned, PHP-FPM runs outside of Apache, so we have another service we can start, stop, reload and restart:

```
# The serivce name is php7.2-fpm,
# not php-fpm
sudo service php7.2-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.

Install PHP-FPM

To install PHP-FPM, we’ll use the package “php-fpm”:

```
sudo apt install -y php-fpm
```

As we mentioned, PHP-FPM runs outside of Apache, so we have another service we can start, stop, reload and restart:

```
sudo service php7.2-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/php/7.2/fpm` directory:

```
$ cd /etc/php/7.2/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:

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/php/7.2/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 `php7.2-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 = /run/php/php7.2-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:

```
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:

```
# Binding to network 192.168.12.*
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:

```
# If you are binding to network 192.168.12.*
listen = 0.0.0.0:9000
```

For each resource pool created, the `listen` directive needs to be set to a unique socket.

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 RAM u\
sed 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!

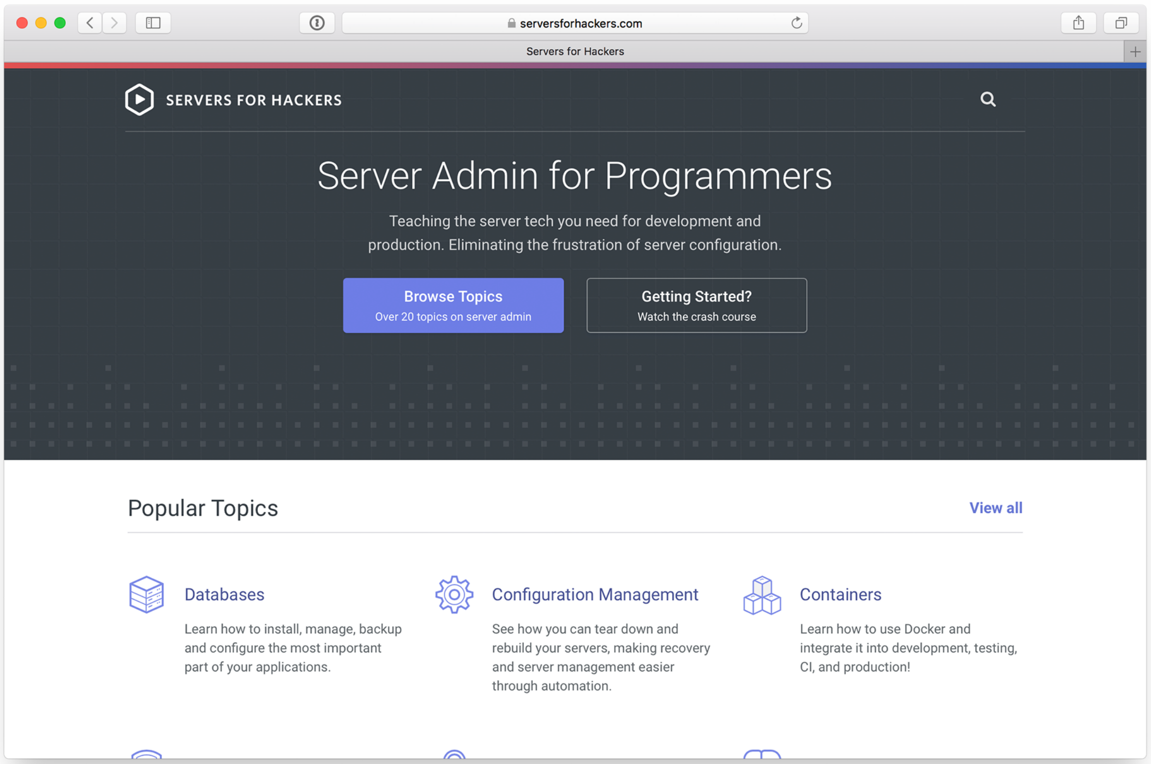
Video Site

A quick note:

Since publishing this book, I've also collected my newsletter editions, articles and videos to https://serversforhackers.com[1].

I'll be continuously adding new videos! 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.



Server Admin for Programmers

Teaching the server tech you need for development and production. Eliminating the frustration of server configuration.

Popular Topics

Databases

Learn how to install, manage, backup and configure the most important part of your applications.

Configuration Management

See how you can tear down and rebuild your servers, making recovery and server management easier through automation.

Containers

Learn how to use Docker and integrate it into development, testing, CI, and production!

Servers for Hackers Video Site

[1] https://serversforhackers.com