This tutorial was written with setting up a Linode server as the goal. You can use this information with any VPS option or on your own machine. If you’re working on a machine that you are not using SSH to access, you can ignore the SSH parts. But it is still good info have.

**Basic Security**

After getting the server turned on, connect to it using either:

ssh root@ipaddress

In a Linux or Mac terminal, or by downloading PuTTY on Windows and connecting via SSH with that.

Now that we are in, before going any further it would be wise to take a few basic steps to harden, or secure, the server. First order of business, updates:

apt update && apt upgrade

Making sure that our server always has the most recent versions of software goes quite a long way in keeping us secure. Next, we’ll create our user, since only having root adds a whole slew of security issues; hackers would be able to log straight into root over SSH giving them access to anything and everything, lack of the sobriety check that having to use sudo gives, and having to setup software with root privileges rather than with a more limited user. To make our new user we can do:

adduser *username*  
and then  
adduser *username* sudo

*username* can be whatever you desire. Using the adduser command actually runs a Perl script that sets up our user with a home directory, a password, and data for contact info if you so desire. Using it the second time adds our new user to the sudo group, allowing us to run commands as root without having to be logged in as root, but avoids allowing the user to have complete access to everything. Since we now have the option to use a non-root user, let’s do so:

su *username*

su means *substitute user*. It is somewhat but not the same as logging out and then in as a new user. Processes start as root continue running and you don’t enter a new shell, but any commands that you run will be done as the new user rather than root.

With all this done, we should now move on to making our SSH connection more secure. Disabling root login and changing which port SSH connections are made on will add a layer of security and stop most automated bots that find websites with insecure SSH rules. We can make these changes in the sshd\_config file, which can be accessed by doing:

sudo nano /etc/ssh/sshd\_config

However, I suggest making a backup first by using the command:

sudo cp /etc/ssh/sshd\_config /etc/ssh/sshd\_config.bak

Just so that if changing a setting breaks everything, we can make it a little less broken by restoring the original configuration.

The parts that interest us are:  
Port 22  
PermitRootLogin yes  
#PermitEmptyPassword no

Set Port to be whatever you desire, just make sure that you will remember it (and don’t worry about suddenly getting kicked off the server by the change, it won’t come into effect until we restart the service). Set PermitRootLogin to no, so that root will only be accessible by doing su root rather than being able to log straight into it (people trying to get in having to crack two passwords is much preferred to having to crack one, no?). Finally, remove the # before PermitEmptyPassword, to make it so that in order to SSH into the server, the user being logged into must have a password. Feel free to play around with other settings in this file, like motd (message of the day, a line of text that will display whenever someone logs into the server over SSH). These changes will come into effect after doing:

sudo service sshd restart

And disconnecting and then reconnecting to the server. Since the SSH port is changed, modify the command you use to connect to be:

ssh *username*@*ipaddress* -p*port*

As far as basic SSH security goes, we are set. There are a few more steps that could be taken, such as setting up SSH Keys to make it so that only someone with a private key that matches the server’s public key can log in over SSH, but I won’t get into that in this tutorial. Perhaps in a future one. Now we’ll get a firewall running.

We have two options. The first is iptables which is the standard for most Linux distributions. The second is UFW, which comes default on Ubuntu and is easily installed on Debian. Of the two, I find UFW to be easier and faster, so that is what we will be using in this tutorial. Getting it installed is as easy as:

sudo apt install ufw

And now we can start adding rules. First, let’s make it so that once the firewall is running we’ll still have SSH access:

sudo ufw allow *PortSetIn sshd\_config*/tcp

Since we are going to be running a web server, we should also allow http and https in and out.

sudo ufw allow http  
sudo ufw allow https

Finally, let’s block everything that isn’t a port we explicitly allow, to minimize holes someone can attempt to enter through, and turn our firewall on.

sudo ufw default deny  
sudo ufw enable

If all was done correctly, you should still be connected the server and your firewall will be all good to go. You can check with:

sudo ufw status

With all this done, let’s get into making us a web server!

**Web Server!**

Step one in making a web server is installing the software needed to make one run. In our case, this will be Apache2 as our server, PHP7.0 as our scripting language, and MariaDB as our database (this is known as a LAMP stack. One can substitute out Apache2 for Nginx and have a LEMP stack. I will make a post on how to make WordPress work with Nginx at a future date). To get all that we will need installed, run:

sudo apt install apache2 mysql-server php7.0 php7.0-mysql php7.0-gd  
sudo systemctl enable apache2.service  
sudo systemctl enable mysql.service

The last two pieces of the install are plugins for PHP to make it play nice with MariaDB and MySQL and also work better with WordPress. And the last two lines tell the system to start Apache and MariaDB at boot.

**Database**

With our toolset installed, let’s first configure MariaDB. The installation comes with a lovely script that you can run by calling:

mysql\_secure\_installation

Which will take of some basic hardening of our database. Say yes to everything it asks, and including setting a secure root password. Three things of note with this, you might have to add sudo to the front of the command, when it asks for current root password it will probably be blank, and it will not like that the password is blank. If you into this issue do the following:

sudo mysql  
and then within your new MySQL shell:  
use mysql;  
update user set password=PASSWORD("*password*") where User='root';  
flush privileges;  
exit

And then try the secure installation script again, this time entering the password you just set. What we just did is go into the main mysql database, change all parts that mention a root password to be something rather than blank, update privileges, and then exit. This makes the secure installation script happy because it now does not have to begin functioning with no password.

Now that we have MariaDB running, let’s make a database for our WordPress install. To start with, let’s get into the MariaDB shell:

mysql -u root -p

This tells MariaDB to open a shell with user root, and then the user does have a password. Sometimes you will need to run sudo with this command, sometimes you will not. Now that we have our MariaDB shell, let’s make a database. I am going to be going the setup for a website that will be called CooperEditing from this point forward. So any time that I mention CooperEditing, change it to match what you desire.

create database CooperEditing;

And now for the sake of security, we will make a user within MariaDB with the ability to edit this database, rather than have root do it.

create user *username*;  
set password for *username*@localhost= PASSWORD("*password*");  
grant all privileges on CooperEditing.\* TO *username*@localhost identified by '*password*';  
flush privileges;  
exit

The first line makes a new MariaDB user, the second sets their password, the third gives them an all access pass to our database, the fourth updates privileges, and the last exits the MariaDB shell. This should be all that we need for our database, so let’s move on to PHP!

**PHP**

Of everything, PHP needs the least configuration to make it work. Actually, it does not need any, but there is one bit that you might want to change in php.ini limits how big of a file can be uploaded to the server. By default it is 2MB. Uploading certain themes or large images and such will require this to be changed. To do so:

sudo nano /etc/php/7.0/apache2.php.ini

The line that reads upload\_max\_filesize (you can use ctrl+w to find this line, similar to ctrl+f in many browsers and graphical text editors) can be changed to adjust how large you want to limit files. I suggest max 5M. If you ever need bigger, adjust it higher, upload the file, and then change it back. Or setup an FTP server.

That’s all that is needed for PHP. So finally, Apache2!

**Apache2**

Apache should not need anything done in its main config file, though if you want to look at it, its location is /etc/apache2/apache2.conf. What we will want to do, is setup a configuration file for our website. We can get this jumpstarted by running:

sudo cp /etc/apache2/sites-available/000-default.conf /etc/apache2/sites-available/CooperEditing.conf  
sudo nano /etc/apache2/sites-available/CooperEditing.conf

There are three lines that interest us and a fourth that we should add:

ServerName www.example.com  
ServerAdmin webmaster@localhost  
DocumentRoot /var/www/html

These should be changed to be in line with what your websites information is. For example, mine ends up being:

ServerAlias cooperediting.com  
ServerName www.cooperediting.com  
ServerAdmin bailey@gingertechnology.net  
DocumentRoot /var/www/CooperEditing

ServerAlias is essentially the website’s domain name, while ServerName is where you can set a subdomain. ServerAdmin is the information on the main administrator of the website (either a local or remote user), and DocumentRoot tells Apache where the files for the website are located. This is all the configuration that will be needed for our website. If you want to host another on the same server, make another config file for it, set the info to be what you need, and you are set.

**Website!**

Let’s start off by making sure we are in our user’s home directory.

cd ~

My reason for doing this is that we will be downloading the files for WordPress, and this makes it easier to keep organized. Now that know we are in our home folder, let’s download the file that we need.

wget http://wordpress.org/latest.tar.gz

Since this is an archive with the most recent build of WordPress in it, we will need to expand it out by running:

tar -xzvf latest.tar.gz

Some text should fly by and you will now have a folder called wordpress in your home directory. We’re almost there! Our next step is to setup the config file our database info from before.

cp wordpress/wp-config-sample/php wordpress/wp-config.php  
nano wordpress/wp-config.php

The lines that we need to change are right at the top.

// \*\* MySQL settings - You can get this info from your web host \*\* //  
/\*\* The name of the database for WordPress \*/  
define('DB\_NAME', 'database\_name\_here');

/\*\* MySQL database username \*/  
define('DB\_USER', 'username\_here');

/\*\* MySQL database password \*/  
define('DB\_PASSWORD', 'password\_here');

Fill in the spots requesting info and save the file. After doing so we need to get the files from our home folder to the DocumentRoot. To do this we can run:

sudo mkdir /var/www/CooperEditing  
sudo cp -R wordpress/\* /var/www/CooperEditing

Transferring the files will end with them not having the right permissions for WordPress to run and Apache2 to display them, so let’s fix that with:

sudo chown -R www-data:www-data /var/www/\*

And if you want your user to be easily able to edit the source files:

sudo adduser *username* www-data

Try going to your server’s IP in your browser, and you should get a WordPress installation page!