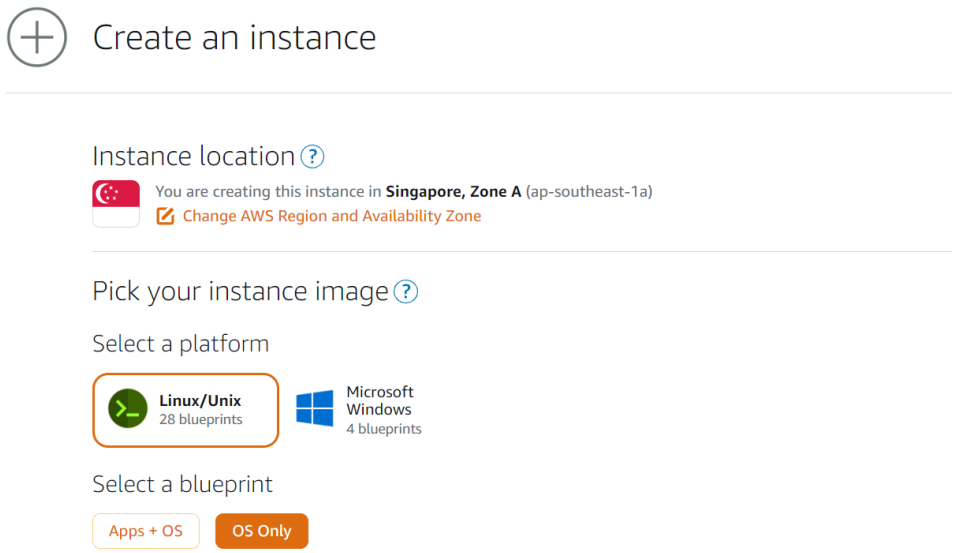
Hosing WordPress on Lightsail Debian Linux with Nginx - PHP8 – MaraiaDB - Cloudflare

Contents

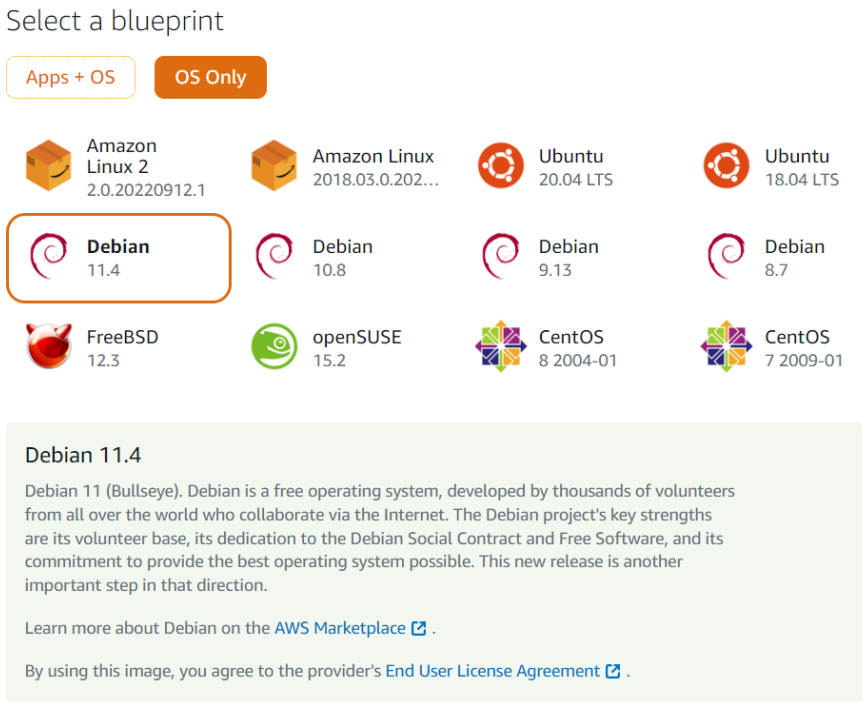
Step 1: Initial Lightsail instance with Debian

In your AWS account, Lightsail Service: create new instance

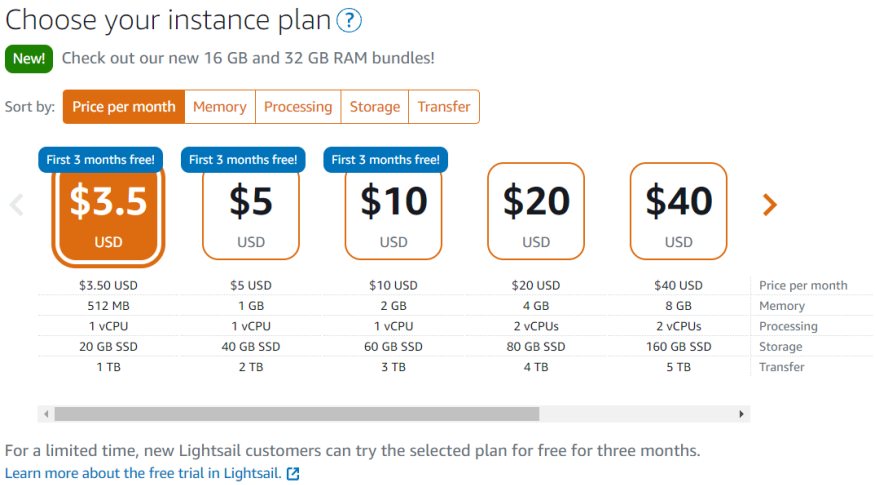
Choose your location and Linux OS only



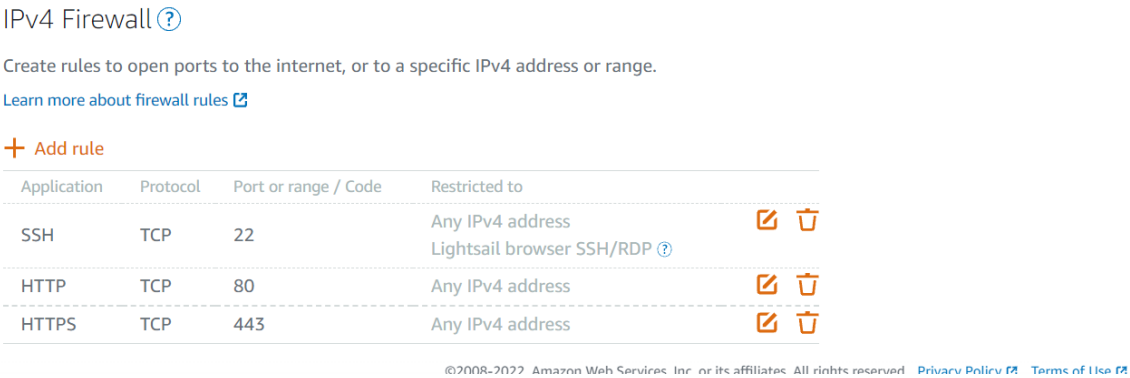
Pick up Debian:



Chose instance plan:

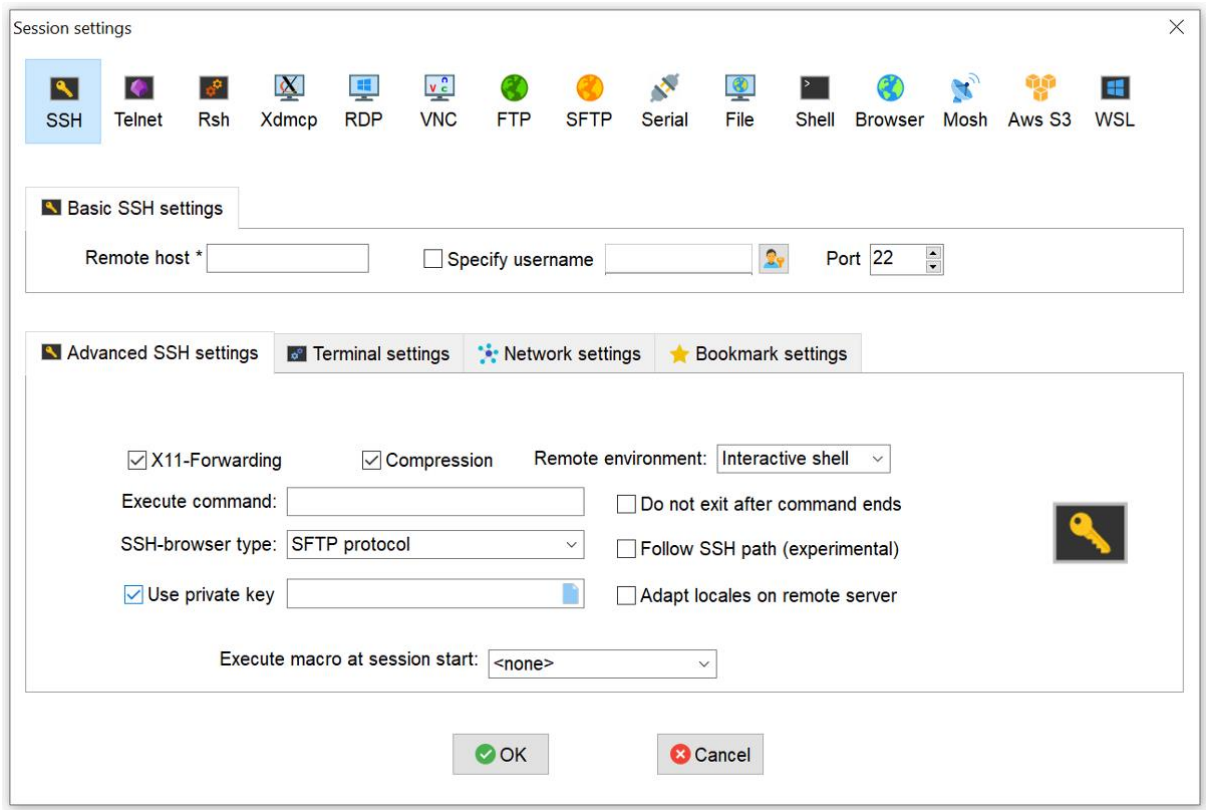


Change the instance Networking firewall to have http and https access for IP4 and IP6



Step 2: SSH to Lightsail instance

Personally I use MobaXterm for SSH connection to server:
add the instance IP address and the default username(admin for Debian) and the pem key file in new session MobaXterm to connect to the server:



Step 3: Installing the Nginx Web Server

In order to serve web pages to your site visitors, we are going to employ Nginx, a popular web server which is well known for its overall performance and stability.

All of the software you will be using for this procedure will come directly from Debian's default package repositories. This means you can use the `apt` package management suite to complete the installation.

Since this is the first time you'll be using `apt` for this session, you should start off by updating your local package index. You can then install the server:

```
sudo apt update

sudo apt install nginx
```

On Debian, Nginx is configured to start running upon installation.

Step 4: Installing PHP

Step 1: Update Debian System

We recommend you perform any installation on an upgraded Debian system. Run the commands provided to upgrade all packages and minor release of your Debian system.

```
sudo apt update

sudo apt -y upgrade
```

OS release can be checked with the command below:

```
$ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

If your kernel was upgraded you can reboot the system to start using it.

```
sudo reboot
```

Step 2: Add Sury APT repository to Debian

The DEB.SURY.ORG repository is a home for packaging various software into Debian and Ubuntu based Linux distributions. It contains the latest binary builds of PHP 8.1. The repository has to be added manually on the system.

Install required temporary packages:

```
sudo apt update
sudo apt install -y lsb-release ca-certificates apt-transport-https software-properties-common gnupg2
```

Add Sury Debian PPA repository to your Debian system.

```
echo "deb https://packages.sury.org/php/ $(lsb_release -sc) main" | sudo tee /etc/apt/sources.list.d/sury-php.list
```

Import packages signing GPG key:

```
curl -fsSL  https://packages.sury.org/php/apt.gpg| sudo gpg --dearmor -o /etc/apt/trusted.gpg.d/sury-keyring.gpg
```

Confirm if the repository is working by downloading package information from all configured sources:

```
$ sudo apt update
Hit:1 http://security.debian.org/debian-security bullseye-security InRelease
Hit:2 http://deb.debian.org/debian bullseye InRelease
Hit:3 http://deb.debian.org/debian bullseye-updates InRelease
Hit:4 http://deb.debian.org/debian bullseye-backports InRelease
Hit:5 https://packages.sury.org/php bullseye InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
All packages are up to date.
```

Step 3: Install PHP 8.1 on Debian

Without Apache2:

```
sudo apt install --no-install-recommends php8.1
```

The `--no-install-recommends` flag will ensure that other packages like the Apache web server are not installed.

With Apache2

Once you've added the repository and is confirmed to be functional, we can proceed with the installation of PHP 8.1 o Debian

```
sudo apt update

sudo apt install php8.1
```

All dependency packages are installed automatically for you. Just hit **y** in your keyboard to proceed:

```
....
Reading state information... Done
The following additional packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils bzip2 libapache2-mod-php8.1 libapr1 libaprutil1 libaprutil1-dbd-sqlite3 libaprutil1-ldap libgdbm-compat4 libjansson4 liblua5.3-0 libperl5.32
  libsodium23 mailcap mime-support perl perl-modules-5.32 php-common php8.1-cli php8.1-common php8.1-opcache php8.1-readline ssl-cert
Suggested packages:
  apache2-doc apache2-suexec-pristine | apache2-suexec-custom www-browser bzip2-doc php-pear perl-doc libterm-readline-gnu-perl | libterm-readline-perl-perl make libtap-harness-archive-perl
The following NEW packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils bzip2 libapache2-mod-php8.1 libapr1 libaprutil1 libaprutil1-dbd-sqlite3 libaprutil1-ldap libgdbm-compat4 libjansson4 liblua5.3-0 libperl5.32
  libsodium23 mailcap mime-support perl perl-modules-5.32 php-common php8.1 php8.1-cli php8.1-common php8.1-opcache php8.1-readline ssl-cert
0 upgraded, 26 newly installed, 0 to remove and 0 not upgraded.
Need to get 14.3 MB of archives.
After this operation, 77.2 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

Installed version of PHP can be checked with the command below after a successful installation

```
$ php -v
PHP 8.1.9 (cli) (built: Aug 15 2022 09:47:52) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.1.9, Copyright (c) Zend Technologies
    with Zend OPcache v8.1.9, Copyright (c), by Zend Technologies
```

Step 4: Install PHP 8.1 Extensions on Debian

PHP 8.1 modules can be installed as packages using below command syntax:

```
sudo apt install php8.1-<extension>
```

Where:

<extension> is replaced with the actual extension name.

Below is an example to install MySQL PHP extension:

```
$ sudo apt install php8.1-mysql

Reading package lists... Done

Building dependency tree... Done

Reading state information... Done

The following NEW packages will be installed:

  php8.1-mysql

0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.

Need to get 116 kB of archives.

After this operation, 466 kB of additional disk space will be used.

Get:1 https://packages.sury.org/php bullseye/main amd64 php8.1-mysql amd64 8.1.0~rc6-3+0~20211120.8+debian11~1.gbp2227b0 [116 kB]

Fetched 116 kB in 0s (1152 kB/s)

Selecting previously unselected package php8.1-mysql.

(Reading database ... 37704 files and directories currently installed.)

Preparing to unpack .../php8.1-mysql_8.1.9-1+0~20220815.24+debian11~1.gbp4d5b5a_amd64.deb ...

Unpacking php8.1-mysql (8.1.9-1+0~20220815.24+debian11~1.gbp4d5b5a) ...

Setting up php8.1-mysql (8.1.9-1+0~20220815.24+debian11~1.gbp4d5b5a) ...


Creating config file /etc/php/8.1/mods-available/mysqlnd.ini with new version


Creating config file /etc/php/8.1/mods-available/mysqli.ini with new version


Creating config file /etc/php/8.1/mods-available/pdo_mysql.ini with new version

Processing triggers for libapache2-mod-php8.1 (8.1.9-1+0~20220815.24+debian11~1.gbp4d5b5a) ...

Processing triggers for php8.1-cli (8.1.9-1+0~20220815.24+debian11~1.gbp4d5b5a) ...
```

To install all commonly used PHP extensions run the following commands a user with sudo privileges:

```
sudo apt install php8.1-
{bcmath,fpm,xml,mysql,zip,intl,ldap,gd,cli,bz2,curl,mbstring,pgsql,opcache,soap,cgi}
```

Continue with the installation:

```
....
uggested packages:
  libgd-tools php-pear
The following NEW packages will be installed:
  fontconfig-config fonts-dejavu-core libdeflate0 libfontconfig1 libgd3 libjbig0 libjpeg62-turbo
libonig5 libpq5 libtiff5 libwebp6 libx11-6 libx11-data libxau6 libxcb1 libxdmcp6 libxpm4 libxslt1.1
  libzip4 php8.1-bcmath php8.1-bz2 php8.1-cgi php8.1-curl php8.1-gd php8.1-intl php8.1-ldap
php8.1-mbstring php8.1-pgsql php8.1-soap php8.1-xml php8.1-zip
0 upgraded, 31 newly installed, 0 to remove and 0 not upgraded.
Need to get 7150 kB of archives.
After this operation, 26.5 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

Check loaded PHP modules using the command:

```
$ php --modules
[PHP Modules]
bcmath
bz2
calendar
Core
ctype
curl
date
dom
exif
FFI
fileinfo
filter
ftp
gd
gettext
hash
iconv
intl
json
ldap
libxml
mbstring
mysqli
mysqlnd
openssl
pcntl
pcre
PDO
pdo_mysql
```

That’s all we had to cover on how to install PHP 8.1 on Debian Linux system.

When you are finished installing the new extensions, you’ll need to restart the PHP-FPM process so that the running PHP processor can leverage the newly installed features:

```
sudo systemctl restart php8.1-fpm.service
```

Step 5: Configuring Nginx

We’ll now make a few minor adjustments to our Nginx server block files. Based on the prerequisite tutorials, you should have a configuration file for your site in the `/etc/nginx/sites-available/` directory configured to respond to your server’s domain name and protected by a TLS/SSL certificate. We’ll use `/etc/nginx/sites-available/your_domain` as an example here, but you should substitute the path to your configuration file where appropriate.

Note: It’s possible you are using the `/etc/nginx/sites-available/default` default configuration (with `/var/www/html` as your web root). This is fine to use if you’re only going to host one website on this server. If not, it’s best to split the necessary configuration into logical chunks, one file per site example of DEFAULT SITE CONFIG

WORKING WITH PHP:

```
# Default server configuration
server {
	listen 80 default_server;
	listen [::]:80 default_server;
	# SSL configuration
	#
	# listen 443 ssl default_server;
	# listen [::]:443 ssl default_server;
	#
	# Note: You should disable gzip for SSL traffic.
	# See: https://bugs.debian.org/773332
	#
	# Read up on ssl_ciphers to ensure a secure configuration.
	# See: https://bugs.debian.org/765782
	#
	# Self signed certs generated by the ssl-cert package
	# Don't use them in a production server!
	#
	# include snippets/snakeoil.conf;

	root /var/www/html;

	# Add index.php to the list if you are using PHP
	index index.html index.htm index.nginx-debian.html;
	server_name _;
	location / {
		# First attempt to serve request as file, then
		# as directory, then fall back to displaying a 404.
		try_files $uri $uri/ =404;
	}

	# pass PHP scripts to FastCGI server
	#
	#location ~ \.php$ {
	#	include snippets/fastcgi-php.conf;
	#
	#	# With php-fpm (or other unix sockets):
	#	fastcgi_pass unix:/run/php/php7.4-fpm.sock;
	#	# With php-cgi (or other tcp sockets):
	#	fastcgi_pass 127.0.0.1:9000;
	#}

	# deny access to .htaccess files, if Apache's document root
	# concurs with nginx's one
	#
	#location ~ /\.ht {
	#	deny all;
	#}
}
```

Additionally, we will use `/var/www/your_domain` as the root directory of our WordPress install. You should use the web root specified in your own configuration.

Open your site's Nginx configuration file with `sudo` privileges to begin:

```
sudo nano /etc/nginx/sites-available/your_domain
```

We need to add a few `location` directives within our main `server` block. After adding SSL certificates your config may have *two* `server` blocks. If so, find the one that contains `root /var/www/your_domain` and your other `location` directives and implement your changes there.

Start by creating exact-matching location blocks for requests to `/favicon.ico` and `/robots.txt`, both of which we do not want to log requests for.

We will use a regular expression location to match any requests for static files. We will again turn off the logging for these requests and will mark them as highly cacheable since these are typically expensive resources to serve. You can adjust this static files list to contain any other file extensions your site may use:

```
server {
    . . .

    location = /favicon.ico { log_not_found off; access_log off; }
    location = /robots.txt { log_not_found off; access_log off; allow all; }
    location ~* \.(css|gif|ico|jpeg|jpg|js|png)$ {
        expires max;
        log_not_found off;
    }
    . . .
}
```

Inside of the existing `location /` block, we need to adjust the `try_files` list so that instead of returning a 404 error as the default option, control is passed to the `index.php` file with the request arguments.

This should look something like this:

```
server {
    . . .
    location / {
        #try_files $uri $uri/ =404;
        try_files $uri $uri/ /index.php$is_args$args;
    }
    . . .
}
```

When you are finished, save and close the file.

File config example :

```
server {
    listen 80;
    listen [::]:80;

    root /var/www/your_domain.com;
    index index.php index.html index.htm;

    server_name your_domain.com www. your_domain.com;

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

    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/var/run/php/php8.1-fpm.sock;
    }
}
```

Manage Nginx:

Now, we can check our configuration for syntax errors by typing:

```
sudo nginx -t
```

If no errors were reported, reload Nginx by typing:

```
sudo systemctl reload nginx
```

Step 6: Install MariaDB

The short version of this installation guide consists of these three steps:

Update your package index using `apt`

Install the `mariadb-server` package using `apt`. The package also pulls in related tools to interact with MariaDB

Run the included `mysql_secure_installation` security script to restrict access to the server

```
sudo apt update

sudo apt install mariadb-server

sudo mysql_secure_installation
```

Creating a Database and User for WordPress

```
sudo mariadb
```

First, we can create a separate database that WordPress can control. You can name this whatever you would like, but we will be using `wordpress` in this guide to keep it simple. You can create the database for WordPress by typing:

```
CREATE DATABASE wordpress DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci;
```

Next, we are going to create a separate MariaDB user account that we will use exclusively to operate on our new database. Creating one-function databases and accounts is a good idea from a management and security standpoint. We will use the name `wordpress_user` in this guide. Feel free to change this if you'd like.

The following command will create this account, set a password, and grant access to the database we created. Remember to choose a strong password for your database user:

```
GRANT ALL ON wordpress.* TO 'wordpress_user'@'localhost' IDENTIFIED BY 'password';
```

You now have a database and a user account, each made specifically for WordPress. We need to flush the privileges so that the current instance of the database server knows about the recent changes we've made:

```
FLUSH PRIVILEGES;
```

Exit out of MariaDB by typing:

```
EXIT;
```

Step 7: Uploading WordPress

Uploading site files :

Assume you developed WordPress website locally in your machine
now will upload the website files and db to the server:

1. zip to the WordPress files to one zip file then upload it to /tmp in the server using MobaXterm
2. export the db from your localhost as sql file

Setting up site files on the web server:

Now in the server :

1. unzip the WordPress file, update the wp_config.php file with the correct db username and password
2. Now, we can copy the entire contents of the directory into our document root. We are using the `-a` flag to make sure our permissions are maintained. We are using a dot at the end of our source directory to indicate that everything within the directory should be copied, including any hidden files:
   ```
   sudo cp -a /tmp/wordpress/. /var/www/your_domain
   ```
3. Now that our files are in place, we'll assign ownership them to the `www-data` user and group. This is the user and group that Nginx runs as, and Nginx will need to be able to read and write WordPress files in order to serve the website and perform automatic updates.

```
sudo chown -R www-data:www-data /var/www/your_domain
```

Our files are now in our server's document root and have the correct ownership, but we still need to complete some more configuration.

Importing Database

Import the db from the exported database sql file to the server MariaDB database: To import an existing dump file into MySQL or MariaDB, you will have to create a new database. This database will hold the imported data:

```
mysql -u username -p new_database < data-dump.sql
```

- `username` is the username you can log in to the database with
- `newdatabase` is the name of the freshly created database
- `data-dump.sql` is the data dump file to be imported, located in the current directory

If the command runs successfully, it won't produce any output. If any errors occur during the process, `mysql` will print them to the terminal instead. To check if the import was successful, log in to the MySQL shell and inspect the data. Selecting the new database with `USE new_database` and then use `SHOW TABLES;` or a similar command to look at some of the data.

Step 8: Setting up the WordPress Configuration File

Next, we need to make a few changes to the main WordPress configuration file.

When we open the file, our first order of business will be to adjust the secret keys to provide some security for our installation. WordPress provides a secure generator for these values so that you do not have to try to come up with good values on your own. These are only used internally, so it won't hurt usability to have complex, secure values here.

To grab secure values from the WordPress secret key generator, type:

```
curl -s https://api.wordpress.org/secret-key/1.1/salt/
```

You will get back unique values that look something like this:

```
Output
define('AUTH_KEY',         '1jl/vqfs<XhdXoAPz9 DO NOT COPY THESE VALUES c_j{iwqD^<+c9.k<J@4H');
define('SECURE_AUTH_KEY',  'E2N-h2]Dcvp+aS/p7X DO NOT COPY THESE VALUES {Ka(f;rv?Pxf})CgLi-3');
define('LOGGED_IN_KEY',    'W(50,{W^,OPB%PB<JF DO NOT COPY THESE VALUES 2;y&,2m%3]R6DUth[;88');
define('NONCE_KEY',        'll,4UC)7ua+8<!4VM+ DO NOT COPY THESE VALUES #`DXF+[$atzM7 o^-C7g');
define('AUTH_SALT',        'koMrurzOA+|L_lG}kf DO NOT COPY THESE VALUES  07VC*Lj*lD&?3w!BT#-');
define('SECURE_AUTH_SALT', 'p32*p,]z%LZ+pAu:VY DO NOT COPY THESE VALUES C-?y+K0DK_+F|0h{!_xY');
define('LOGGED_IN_SALT',   'i^/G2W7!-1H2OQ+t$3 DO NOT COPY THESE VALUES t6**bRVFSD[Hi])-qS`|');
define('NONCE_SALT',       'Q6]U:K?j4L%Z]}h^q7 DO NOT COPY THESE VALUES 1% ^qUswWgn+6&xqHN&%');
```

These are configuration lines that we can paste directly in our configuration file to set secure keys. Copy the output you received now.

Now, open the WordPress configuration file:

```
nano /var/www/your_domain/wp-config.php
```

Find the section that contains the dummy values for those settings. It will look something like this:

```
. . .

define('AUTH_KEY',         'put your unique phrase here');
define('SECURE_AUTH_KEY',  'put your unique phrase here');
define('LOGGED_IN_KEY',    'put your unique phrase here');
define('NONCE_KEY',        'put your unique phrase here');
define('AUTH_SALT',        'put your unique phrase here');
define('SECURE_AUTH_SALT', 'put your unique phrase here');
define('LOGGED_IN_SALT',   'put your unique phrase here');
define('NONCE_SALT',       'put your unique phrase here');

. . .
```

Delete those lines and paste in the values you copied from the command line:

```
. . .

define('AUTH_KEY',         'VALUES COPIED FROM THE COMMAND LINE');
define('SECURE_AUTH_KEY',  'VALUES COPIED FROM THE COMMAND LINE');
define('LOGGED_IN_KEY',    'VALUES COPIED FROM THE COMMAND LINE');
define('NONCE_KEY',        'VALUES COPIED FROM THE COMMAND LINE');
define('AUTH_SALT',        'VALUES COPIED FROM THE COMMAND LINE');
define('SECURE_AUTH_SALT', 'VALUES COPIED FROM THE COMMAND LINE');
define('LOGGED_IN_SALT',   'VALUES COPIED FROM THE COMMAND LINE');
define('NONCE_SALT',       'VALUES COPIED FROM THE COMMAND LINE');

. . .
```

Next, we need to modify some of the database connection settings at the beginning of the file. You need to adjust the database name, the database user, and the associated password that we configured within MariaDB.

The other change we need to make is to set the method that WordPress should use to write to the filesystem. Since we've given the web server permission to write where it needs to, we can explicitly set the filesystem method to "direct". Failure to set this with our current settings would result in

WordPress prompting for FTP credentials when we perform some actions. This setting can be added below the database connection settings, or anywhere else in the file:

```
. . .
define('DB_NAME', 'wordpress');


/** MySQL database username */
define('DB_USER', 'wordpress_user');
/** MySQL database password */
define('DB_PASSWORD', 'password');
. . .
define('FS_METHOD', 'direct');
```

Update the domain: siteurl and home:

```
UPDATE `wp_options` SET `option_value` = 'https://your-domain.com' WHERE `wp_options`.`option_id` = 2;
UPDATE `wp_options` SET `option_value` = 'https://your-domain.com' WHERE `wp_options`.`option_id` = 1;
```

Step 9: Secure Nginx with Let's Encrypt

Step 1 — Installing Certbot

The first step to using Let's Encrypt to obtain an SSL certificate is to install the Certbot software on your server.

Install Certbot and its Nginx plugin with `apt`:

```
sudo apt install certbot python3-certbot-nginx
```

Certbot is now ready to use, but in order for it to automatically configure SSL for Nginx, we need to verify some of Nginx's configuration.

Step 2 — Confirming Nginx's Configuration

Certbot needs to be able to find the correct `server` block in your Nginx configuration for it to be able to automatically configure SSL. Specifically, it does this by looking for a `server_name` directive that matches the domain you request a certificate for.

If you followed the server block set up step in the Nginx installation tutorial, you should have a server block for your domain at `/etc/nginx/sites-available/example.com` with the `server_name` directive already set appropriately.

To check, open the configuration file for your domain using `nano` or your favorite text editor:

```
sudo nano /etc/nginx/sites-available/example.com
```

Find the existing `server_name` line. It should look like this:

```
...
server_name example.com www.example.com;
...
```

If it does, exit your editor and move on to the next step.

If it doesn't, update it to match. Then save the file, quit your editor, and verify the syntax of your configuration edits:

```
sudo nginx -t
```

If you get an error, reopen the server block file and check for any typos or missing characters. Once your configuration file's syntax is correct, reload Nginx to load the new configuration:

```
sudo systemctl reload nginx
```

Step 3— Verifying Certbot Auto-Renewal

Let's Encrypt's certificates are only valid for ninety days. This is to encourage users to automate their certificate renewal process. The `certbot` package we installed takes care of this for us by adding a systemd timer that will run twice a day and automatically renew any certificate that's within thirty days of expiration.

You can query the status of the timer with `systemctl`:

```
sudo systemctl status certbot.timer
```

```
Output
● certbot.timer - Run certbot twice daily
     Loaded: loaded (/lib/systemd/system/certbot.timer; enabled; vendor preset: enabled)
     Active: active (waiting) since Mon 2022-08-08 19:05:35 UTC; 11s ago
    Trigger: Tue 2022-08-09 07:22:51 UTC; 12h left
   Triggers: ● certbot.service
```

To test the renewal process, you can do a dry run with `certbot`:

```
sudo certbot renew --dry-run
```

If you see no errors, you're all set. When necessary, Certbot will renew your certificates and reload Nginx to pick up the changes. If the automated renewal process ever fails, Let's Encrypt will send a message to the email you specified, warning you when your certificate is about to expire.

Step 10: How To Implement Browser Caching with Nginx's header Module

Creating Test Files

In this step, we will create several test files in the default Nginx directory. We'll use these files later to check Nginx's default behavior and then to test that browser caching is working.

To infer what kind of file is served over the network, Nginx does not analyze the file contents; that would be prohibitively slow. Instead, it looks up the file extension to determine the file's *MIME type*, which denotes its purpose.

Because of this behavior, the content of our test files is irrelevant. By naming the files appropriately, we can trick Nginx into thinking that, for example, one entirely empty file is an image and another is a stylesheet.

Create a file named `test.html` in the default Nginx directory using `truncate`. This extension denotes that it's an HTML page:

```
sudo truncate -s 1k /var/www/html/test.html
```

Let's create a few more test files in the same manner: one `jpg` image file, one `css` stylesheet, and one `js` JavaScript file:

```
sudo truncate -s 1k /var/www/html/test.jpg
sudo truncate -s 1k /var/www/html/test.css
sudo truncate -s 1k /var/www/html/test.js
```

The next step is to check how Nginx behaves with respect to sending caching control headers on a fresh installation with the files we have just created.

Checking the Default Behavior

By default, all files will have the same default caching behavior. To explore this, we'll use the HTML file we created in step 1, but you can run these tests with any example files.

So, let's check if `test.html` is served with any information regarding how long the browser should cache the response. The following command requests a file from our local Nginx server and shows the response headers:

```
curl -I http://localhost/test.html
```

You will see several HTTP response headers:

```
Output: Nginx response headers
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Tue, 02 Feb 2021 19:03:21 GMT
Content-Type: text/html
Content-Length: 1024
Last-Modified: Tue, 02 Feb 2021 19:02:58 GMT
Connection: keep-alive
ETag: "6019a1e2-400"
Accept-Ranges: bytes
```

In the second to last line you will find the `ETag` header, which contains a unique identifier for this particular revision of the requested file. If you execute the previous `curl` command repeatedly, you will find the exact same `ETag` value.

When using a web browser, the `ETag` value is stored and sent back to the server with the `If-None-Match` request header when the browser wants to request the same file again — for example, when refreshing the page.

We can simulate this on the command line with the following command. Make sure you change the `ETag` value in this command to match the `ETag` value in your previous output:

```
curl -I -H 'If-None-Match: "6019a1e2-400"' http://localhost/test.html
```

The response will now be different:

```
Output: Nginx response headers
HTTP/1.1 304 Not Modified
Server: nginx/1.18.0 (Ubuntu)
Date: Tue, 02 Feb 2021 19:04:09 GMT
Last-Modified: Tue, 02 Feb 2021 19:02:58 GMT
Connection: keep-alive
ETag: "6019a1e2-400"
```

This time, Nginx will respond with **304 Not Modified**. It won't send the file over the network again; instead, it will tell the browser that it can reuse the file it already has downloaded locally.

This is useful because it reduces network traffic, but it's not good enough to achieve good caching performance. The problem with `ETag` is that browsers always send a request to the server asking if it can reuse its cached file. Even though the server responds with a 304 instead of sending the file again, it still takes time to make the request and receive the response.

In the next step, we will use the headers module to append caching control information. This will make the browser cache some files locally without explicitly asking the server if its fine to do so.

Configuring Cache-Control and Expires Headers

In addition to the `ETag` file validation header, there are two caching control response headers: `Cache-Control` and `Expires`. `Cache-Control` is the newer version, with more options than `Expires` and is generally more useful if you want finer control over your caching behavior.

If these headers are set, they can tell the browser that the requested file can be kept locally for a certain amount of time (including forever) without requesting it again. If the headers are not set, browsers will always request the file from the server, expecting either **200 OK** or **304 Not Modified** responses.

We can use the header module to set these HTTP headers. The header module is a core Nginx module, which means it doesn't need to be installed separately to be used.

To add the header module, open the default Nginx configuration file in `nano` or your favorite text editor:

```
sudo nano /etc/nginx/sites-available/default
```

Find the `server` configuration block:

```
. . .
# Default server configuration
#

server {
    listen 80 default_server;
    listen [::]:80 default_server;

. . .
```

Add the following two new sections here: one before the `server` block, to define how long to cache different file types, and one inside it, to set the caching headers appropriately:

```
. . .
# Default server configuration
#

# Expires map
map $sent_http_content_type $expires {
    default                 off;
    text/html               epoch;

    text/css                max;
    text/xml                max;
    image/gif               max;
    image/jpeg              max;
    application/javascript  max;
    image/avif              max;
    image/png               max;
    image/svg+xml           max;
    image/tiff              max;
    image/vnd.wap.wbmp      max;
    image/webp              max;
    image/x-icon            max;
    image/x-jng             max;
    image/x-ms-bmp          max;
    font/woff               max;
    font/woff2              max;
    application/pdf         max;
    application/zip         max;
    audio/midi              max;
    audio/mpeg              max;
    audio/ogg               max;
    audio/x-m4a             max;
    audio/x-realaudio       max;
    video/3gpp              max;
    video/mp2t              max;
    video/mp4               max;
    video/mpeg              max;
    video/quicktime         max;
    video/webm              max;
    video/x-flv             max;
    video/x-m4v             max;
    video/x-mng             max;
    video/x-ms-asf          max;
    video/x-ms-wmv          max;
    video/x-msvideo         max;
}

server {
    listen 80 default_server;
    listen [::]:80 default_server;

    expires $expires;
```

The section before the `server` block is a new `map` block that defines the mapping between the file type and how long that kind of file should be cached.

We're using several different settings in this map:

- The default value is set to `off`, which will not add any caching control headers. It's a safe bet for the content, we have no particular requirements on how the cache should work.
- For `text/html`, we set the value to `epoch`. This is a special value that results explicitly in no caching, which forces the browser to always ask if the website itself is up to date.
- For `text/css` and `application/javascript`, which are stylesheets and JavaScript files, we set the value to `max`. This means the browser will cache these files for as long as possible, reducing the number of requests considerably given that there are typically many of these files.
- The last two settings are for `~image/` and `~font/`, which are regular expressions that will match all file types containing `image/` or `font/` in their *MIME type* name (like `image/jpg`, `image/png` or `font/woff2`). Like stylesheets, both pictures and web fonts on websites can be safely cached to speed up page-loading times, so we set this to `max` as well.

Note: These are just a few examples of the most common MIME types used on websites. You can get acquainted with the more extensive list of such types on Common MIME types site and add others to the map that you might find useful in your case.

Inside the server block, the `expires` directive (a part of the headers module) sets the caching control headers. It uses the value from the `$expires` variable set in the map. This way, the resulting headers will be different depending on the file type.

Save and close the file to exit.

To enable the new configuration, restart Nginx:

```
sudo systemctl restart nginx
```

Next, let's make sure our new configuration works.

Testing Browser Caching

Execute the same request as before for the test HTML file:

```
curl -I http://localhost/test.html
```

This time the response will be different. You will see two additional HTTP response headers:

```
Output
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Tue, 02 Feb 2021 19:10:13 GMT
Content-Type: text/html
Content-Length: 1024
Last-Modified: Tue, 02 Feb 2021 19:02:58 GMT
Connection: keep-alive
ETag: "6019a1e2-400"
Expires: Thu, 01 Jan 1970 00:00:01 GMT
Cache-Control: no-cache
Accept-Ranges: bytes
```

The `Expires` header shows a date in the past and `Cache-Control` is set with `no-cache`, which tells the browser to always ask the server if there is a newer version of the file (using the `ETag` header, like before).

You'll find a difference in response with the test image file:

```
curl -I http://localhost/test.jpg
```

Note the new output:

```
Output
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Tue, 02 Feb 2021 19:10:42 GMT
Content-Type: image/jpeg
Content-Length: 1024
Last-Modified: Tue, 02 Feb 2021 19:03:02 GMT
Connection: keep-alive
ETag: "6019a1e6-400"
Expires: Thu, 31 Dec 2037 23:55:55 GMT
Cache-Control: max-age=315360000
Accept-Ranges: bytes
```

In this case, `Expires` shows the date in the distant future, and `Cache-Control` contains `max-age` information, which tells the browser how long it can cache the file in seconds. This tells the browser to cache the downloaded image for as long as it can, so any subsequent appearances of this image will use local cache and not send a request to the server at all.

The result should be similar for both `test.js` and `test.css`, as both JavaScript and stylesheet files are set with caching headers too.

This means the cache control headers have been configured properly and your website will benefit from the performance gain and less server requests due to browser caching. You should customize the caching settings based on your website's content, but the defaults in this article are a reasonable place to start.

Step 11: Nginx Logs:

nginx logs are stored in:

```
/var/log/nginx/error.log
```

Step 12: Enabling Nginx mod_rewrite

Nginx does not support .htaccess files.

So to enable rewrite mode:

replace this line in the configuration file, inside the location block

```
try_files $uri $uri/ =404;
```

with this new line:

```
try_files $uri $uri/ /index.php?$args;
```

Finally, it will look like this:

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

Step 13: Optimize Nginx

Since the Mem is only 512MB, then it is better to optimize Nginx server to avoid 502 bad gateway error or NGINX: upstream timed out and so on.

for this optimization:

1. will set the pm.max_children = 1 , and this is why I did that :
   Check how much RAM you have to spare (perhaps stop php-fpm to calculate this) and how much memory a php-fpm process typically takes for your webapp (look at top or ps) and then set pm.max_children = available RAM / php-fpm process RAM , otherwise you'll run out of memory.
2. Optimize timeout: This happens because your upstream takes too long to answer the request and NGINX thinks the upstream already failed in processing the request, so it responds with an error. Just include and increase proxy_read_timeout in location config block. Same thing happened to me and I used 1 hour timeout for an internal app at work:

   ```
   proxy_read_timeout 3600;
   ```

   With this, NGINX will wait for an hour (3600s) for its upstream to return something.

   ```
   location ~ \.php$ {
   ....
        proxy_read_timeout        3600;
   ....
     }
   ```

or this:

So i have to adjust the fastcgi_read_timeout in my server configuration

```
location ~ \.php$ {
   fastcgi_read_timeout 240;
   ...
}
```

3. It is recommend to have a look at on this settings as well :

```
# keepalive settings

    keepalive_requests 10000;

    keepalive_timeout 6;
```

inside the file: **/etc/nginx/nginx.conf**

Step 14: Setup Cloudflare

https://www.youtube.com/watch?v=C5_uK44XSqY

The previous YouTube video will guide you how to:

1. how to install Cloudflare by changing the nameservers (examples of how to do it from NameCheap & GoDaddy).
2. adjust the most important settings in Cloudflare to ensure your website is optimized for speed and security.
3. show you a cool WP plugin to control everything from your WordPress dashboard.
4. At the end, show you real user data and how much Cloudflare improved page speed.

Note: this is the page rules:

Page Rule Cloud Flare:

Your-domain.com/wp-login.php* : security level: high

Your-domain.com/wp-admin* : security level: high - cache level: bypass - disable performance - disable apps

http://* Your-domain.com/* : always use https - order first

References:

- https://www.digitalocean.com/community/tutorials/how-to-install-mariadb-on-debian-11
- https://computingforgeeks.com/how-to-install-php-on-debian-linux-2/

- https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-on-debian-11
- https://www.digitalocean.com/community/tutorials/how-to-import-and-export-databases-in-mysql-or-mariadb
- https://www.digitalocean.com/community/tutorials/how-to-install-linux-nginx-mariadb-php-lemp-stack-on-debian-10
- https://github.com/nginx/nginx/blob/master/conf/mime.types
- https://www.digitalocean.com/community/tutorials/how-to-implement-browser-caching-with-nginx-s-header-module-on-ubuntu-20-04
- https://kitcharoenp.github.io/nginx/2021/10/21/upstream-timed-out.html
- https://linuxhint.com/what-is-keepalive-in-nginx/
- https://www.youtube.com/watch?v=C5_uK44XSqY

Powered By:
Mo