// Tutorial //

How To Install WordPress With Docker Compose

Updated on January 30, 2024

Docker MySQL Nginx Let's Encrypt WordPress

By Kathleen Juell

English

Introduction

WordPress is a free and open-source Content Management System (CMS) built on a MySQL database with PHP processing. Thanks to its extensible plugin architecture and templating system, most of its administration can be done through the web interface. This is a reason why WordPress is a popular choice when creating different types of websites, from blogs to product pages to eCommerce sites.

Running WordPress typically involves installing a LAMP (Linux, Apache, MySQL, and PHP) or LEMP (Linux, Nginx, MySQL, and PHP) stack, which can be time-consuming. However, by using tools like Docker and Docker Compose, you can streamline the process of setting up your preferred stack and installing WordPress. Instead of installing individual components by hand, you can use *images*, which standardize things like libraries, configuration files, and environment variables. Then, run these images in *containers*, isolated processes that run on a shared operating system. Additionally, by using Compose, you can coordinate multiple containers — for example, an application and database — to communicate with one another.

In this tutorial, you will build a multi-container WordPress installation. Your containers will include a MySQL database, an Nginx web server, and WordPress itself. You will also secure your installation by obtaining TLS/SSL certificates with Let's Encrypt for the domain you want associated with your site. Finally, you will set up a `cron` job to renew your certificates so that your domain remains secure.

Prerequisites

If you are using Ubuntu version 16.04 or below, we recommend you upgrade to a more latest version since Ubuntu no longer provides support for these versions. This collection of guides will help you in upgrading your Ubuntu version.

To follow this tutorial, you will need:

- A server running Ubuntu, along with a non-root user with `sudo` privileges and an active firewall. For guidance on how to set these up, please choose your distribution from this list and follow our Initial Server Setup Guide.
- Docker installed on your server, following **Steps 1 and 2** of "How To Install and Use Docker on Ubuntu" 22.04 / 20.04 / 18.04.
- Docker Compose installed on your server, following **Step 1** of "How To Install Docker Compose on Ubuntu" 22.04 / 20.04 / 18.04.
- A registered domain name. This tutorial will use **your_domain** throughout. You can get one for free at Freenom, or use the domain registrar of your choice.
- Both of the following DNS records set up for your server. You can follow this introduction to DigitalOcean DNS for details on how to add them to a DigitalOcean account:
    - An A record with `your_domain` pointing to your server's public IP address.
    - An A record with `www.your_domain` pointing to your server's public IP address.

Once you have everything set up, you're ready to begin the first step.

Installing WordPress with Docker Compose

1. Define the Web Server Configuration
2. Define Environmental Variables
3. Define Services in Docker Compose
4. Obtain SSL Certificate
5. Modify Web Server Configuration
6. Complete Install via Web Interface
7. Enable SSL to Renew Automatically

Step 1 — Defining the Web Server Configuration

Before running any containers, your first step is to define the configuration for your Nginx web server. Your configuration file will include some WordPress-specific location blocks, along with a location block to direct Let's Encrypt verification requests to the Certbot client for automated certificate renewals.

First, create a project directory for your WordPress setup. In this example, it is called `wordpress`. You can name this directory differently if you'd like to:

```
mkdir wordpress
```
Copy

Then navigate to the directory:

```
cd wordpress
```
Copy

Next, make a directory for the configuration file:

```
mkdir nginx-conf
```
Copy

Open the file with `nano` or your favorite editor:

```
nano nginx-conf/nginx.conf
```
Copy

In this file, add a server block with directives for your server name and document root, and location blocks to direct the Certbot client's request for certificates, PHP processing, and static asset requests.

Add the following code into the file. Be sure to replace `your_domain` with your own domain name:

~/wordpress/nginx-conf/nginx.conf

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

        server_name your_domain www.your_domain;

        index index.php index.html index.htm;

        root /var/www/html;

        location ~ /.well-known/acme-challenge {
                allow all;
                root /var/www/html;
        }

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

```
        location ~ \.php$ {
                try_files $uri =404;
                fastcgi_split_path_info ^(.+\.php)(/.+)$;
                fastcgi_pass wordpress:9000;
                fastcgi_index index.php;
                include fastcgi_params;
                fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
                fastcgi_param PATH_INFO $fastcgi_path_info;
        }

        location ~ /\.ht {
                deny all;
        }

        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;
        }
}
```

Our server block includes the following information:

Directives:

- `listen`: This tells Nginx to listen on port `80`, which will allow you to use Certbot's webroot plugin for your certificate requests. Note that you are *not* including port `443` yet — you will update your configuration to include SSL once you have successfully obtained your certificates.
- `server_name`: This defines your server name and the server block that should be used for requests to your server. Be sure to replace `your_domain` in this line with your own domain name.
- `index`: This directive defines the files that will be used as indexes when processing requests to your server. You modified the default order of priority here, moving `index.php` in front of `index.html` so that Nginx prioritizes files called `index.php` when possible.
- `root`: This directive names the **root** directory for requests to your server. This directory, `/var/www/html`, is created as a mount point at build time by instructions in your WordPress Dockerfile. These Dockerfile instructions also ensure that the files from the WordPress release are mounted to this volume.

Location Blocks:

- `location ~ /.well-known/acme-challenge`: This location block will handle requests to the `.well-known` directory, where Certbot will place a temporary file to validate that the DNS for your domain

resolves to your server. With this configuration in place, you will be able to use Certbot's webroot plugin to obtain certificates for your domain.

- `location /`: In this location block, a `try_files` directive is used to check for files that match individual URI requests. Instead of returning a **404 Not Found** status as a default, however, you'll pass control to WordPress's `index.php` file with the request arguments.
- `location ~ \.php$`: This location block will handle PHP processing and proxy these requests to your `wordpress` container. Because your WordPress Docker image will be based on the `php:fpm` image, you will also include configuration options that are specific to the FastCGI protocol in this block. Nginx requires an independent PHP processor for PHP requests. In this case, these requests will be handled by the `php-fpm` processor that's included with the `php:fpm` image. Additionally, this location block includes FastCGI-specific directives, variables, and options that will proxy requests to the WordPress application running in your `wordpress` container, set the preferred index for the parsed request URI, and parse URI requests.
- `location ~ /\.ht`: This block will handle `.htaccess` files since Nginx won't serve them. The `deny_all` directive ensures that `.htaccess` files will never be served to users.
- `location = /favicon.ico`, `location = /robots.txt`: These blocks ensure that requests to `/favicon.ico` and `/robots.txt` will not be logged.
- `location ~* \.(css|gif|ico|jpeg|jpg|js|png)$`: This block turns off logging for static asset requests and ensures that these assets are highly cacheable, as they are typically expensive to serve.

For more information about FastCGI proxying, read Understanding and Implementing FastCGI Proxying in Nginx. For information about server and location blocks, check out Understanding Nginx Server and Location Block Selection Algorithms.

Save and close the file when you are finished editing. If you used `nano`, do so by pressing `CTRL+X`, `Y`, then `ENTER`.

With your Nginx configuration in place, you can move on to creating environment variables to pass to your application and database containers at runtime.

Step 2 — Defining Environment Variables

Your database and WordPress application containers will need access to certain environment variables at runtime in order for your application data to persist and be accessible to your application. These variables include both sensitive and non-sensitive information: sensitive values for your MySQL **root** password and application database user and password, and non-sensitive information for your application database name and host.

Rather than setting all of these values in your Docker Compose file — the main file that contains information about how your containers will run — set the sensitive values in an `.env` file and restrict its circulation. This will prevent these values from copying over to your project repositories and being exposed publicly.

In your main project directory, `~/wordpress`, open a file called `.env`:

```
nano .env
```

Copy

The confidential values that you set in this file include a password for the MySQL **root** user, and a username and password that WordPress will use to access the database.

Add the following variable names and values to the file. Remember to supply **your own values** here for each variable:

~/wordpress/.env

```
MYSQL_ROOT_PASSWORD=your_root_password
MYSQL_USER=your_wordpress_database_user
MYSQL_PASSWORD=your_wordpress_database_password
```

Included is a password for the **root** administrative account, as well as your preferred username and password for your application database.

Save and close the file when you are finished editing.

Because your `.env` file contains sensitive information, you want to ensure that it is included in your project's `.gitignore` and `.dockerignore` files. This tells Git and Docker what files **not** to copy to your Git repositories and Docker images, respectively.

If you plan to work with Git for version control, initialize your current working directory as a repository with `git init`:

```
git init
```

Copy

Then create and open a `.gitignore` file:

```
nano .gitignore
```

Copy

Add `.env` to the file:

~/wordpress/.gitignore

```
.env
```

Save and close the file when you are finished editing.

Likewise, it's a good precaution to add `.env` to a `.dockerignore` file, so that it doesn't end up on your containers when you are using this directory as your build context.

Open the file:

```
nano .dockerignore
```

Copy

Add `.env` to the file:

~/wordpress/.dockerignore

```
.env
```

Below this, you can optionally add files and directories associated with your application's development:

~/wordpress/.dockerignore

```
.env
.git
docker-compose.yml
.dockerignore
```

Save and close the file when you are finished.

With your sensitive information in place, you can now move on to defining your services in a `docker-compose.yml` file.

Step 3 — Defining Services with Docker Compose

Your `docker-compose.yml` file will contain the service definitions for your setup. A *service* in Compose is a running container, and service definitions specify information about how each container will run.

Using Compose, you can define different services to run multi-container applications since Compose allows you to link these services together with shared networks and volumes. This will be helpful for your current setup since you will create different containers for your database, WordPress application, and web server. You will also create a container to run the Certbot client to obtain certificates for your webserver.

To begin, create and open the `docker-compose.yml` file:

```
nano docker-compose.yml
```

Copy

Add the following code to define your Compose file version and `db` database service:

~/wordpress/docker-compose.yml

```
version: '3'

services:
  db:
    image: mysql:8.0
    container_name: db
    restart: unless-stopped
    env_file: .env
    environment:
      - MYSQL_DATABASE=wordpress
    volumes:
      - dbdata:/var/lib/mysql
    command: '--default-authentication-plugin=mysql_native_password'
    networks:
      - app-network
```

Copy

The `db` service definition contains the following options:

- `image`: This tells Compose what image to pull to create the container. You are pinning the `mysql:8.0` image here to avoid future conflicts as the `mysql:latest` image continues to be updated. For more information about version pinning and avoiding dependency conflicts, read the Docker documentation on Dockerfile best practices.
- `container_name`: This specifies a name for the container.
- `restart`: This defines the container restart policy. The default is `no`, but you have set the container to restart unless it is stopped manually.
- `env_file`: This option tells Compose that you would like to add environment variables from a file called `.env`, located in your build context. In this case, the build context is your current directory.
- `environment`: This option allows you to add additional environment variables, beyond those defined in your `.env` file. You will set the `MYSQL_DATABASE` variable equal to `wordpress` to provide a name for your application database. Because this is non-sensitive information, you can include it directly in the `docker-compose.yml` file.
- `volumes`: Here, you're mounting a named volume called `dbdata` to the `/var/lib/mysql` directory on the container. This is the standard data directory for MySQL on most distributions.
- `command`: This option specifies a command to override the default CMD instruction for the image. In this particular case, you will add an option to the Docker image's standard `mysqld` command, which starts the MySQL server on the container. This option, `--default-authentication-plugin=mysql_native_password`, sets the `--default-authentication-plugin` system variable to `mysql_native_password`, specifying which authentication mechanism should govern new authentication requests to the server. Since PHP and therefore your WordPress image won't support MySQL's newer authentication default, you must make this adjustment in order to authenticate your application database user.
- `networks`: This specifies that your application service will join the `app-network` network, which you will define at the bottom of the file.

Next, below your `db` service definition, add the definition for your `wordpress` application service:

~/wordpress/docker-compose.yml

```
...
  wordpress:
    depends_on:
      - db
    image: wordpress:5.1.1-fpm-alpine
    container_name: wordpress
    restart: unless-stopped
    env_file: .env
    environment:
      - WORDPRESS_DB_HOST=db:3306
      - WORDPRESS_DB_USER=$MYSQL_USER
      - WORDPRESS_DB_PASSWORD=$MYSQL_PASSWORD
      - WORDPRESS_DB_NAME=wordpress
    volumes:
      - wordpress:/var/www/html
    networks:
      - app-network
```

Copy

In this service definition, you're naming your container and defining a restart policy, as you did with the `db` service. You're also adding some options specific to this container:

- `depends_on`: This option ensures that your containers will start in order of dependency, with the `wordpress` container starting after the `db` container. Your WordPress application relies on the existence of your application database and user, so expressing this order of dependency will enable your application to start properly.
- `image`: For this setup, you are using the `5.1.1-fpm-alpine` WordPress image. As discussed in Step 1, using this image ensures that your application will have the `php-fpm` processor that Nginx requires to handle PHP processing. This is also an `alpine` image, derived from the Alpine Linux project, which will help keep your overall image size down. For more information about the benefits and drawbacks of using `alpine` images and whether or not this makes sense for your application, review the full discussion under the **Image Variants** section of the Docker Hub WordPress image page.
- `env_file`: Again, you specify that you want to pull values from your `.env` file, since this is where you defined your application database user and password.
- `environment`: Here, you're using the values you defined in your `.env` file, but are assigning them to the variable names that the WordPress image expects: `WORDPRESS_DB_USER` and `WORDPRESS_DB_PASSWORD`. You're also defining a `WORDPRESS_DB_HOST`, which will be the MySQL server running on the `db` container that's accessible on MySQL's default port, `3306`. Your `WORDPRESS_DB_NAME` will be the same value you specified in the MySQL service definition for your `MYSQL_DATABASE`: `wordpress`.
- `volumes`: You are mounting a named volume called `wordpress` to the `/var/www/html` mountpoint created by the WordPress image. Using a named volume in this way will allow you to share your application code with other containers.
- `networks`: You're also adding the `wordpress` container to the `app-network` network.

Next, below the `wordpress` application service definition, add the following definition for your `webserver` Nginx service:

~/wordpress/docker-compose.yml

```
...
  webserver:
    depends_on:
      - wordpress
    image: nginx:1.15.12-alpine
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - wordpress:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
    networks:
      - app-network
```

Copy

Here, you're naming your container and making it dependent on the `wordpress` container in starting order. You're also using an `alpine` image — the `1.15.12-alpine` Nginx image.

This service definition also includes the following options:

- `ports`: This exposes port `80` to enable the configuration options you defined in your `nginx.conf` file in Step 1.
- `volumes`: Here, you are defining a combination of named volumes and bind mounts:
  - `wordpress:/var/www/html`: This will mount your WordPress application code to the `/var/www/html` directory, the directory you set as the `root` in your Nginx server block.
  - `./nginx-conf:/etc/nginx/conf.d`: This will bind mount the Nginx configuration directory on the host to the relevant directory on the container, ensuring that any changes you make to files on the host will be reflected in the container.
  - `certbot-etc:/etc/letsencrypt`: This will mount the relevant Let's Encrypt certificates and keys for your domain to the appropriate directory on the container.

You've also added this container to the `app-network` network.

Finally, below your `webserver` definition, add your last service definition for the `certbot` service. Be sure to replace the email address and domain names listed here with your own information:

~/wordpress/docker-compose.yml

```
  certbot:
    depends_on:
```

Copy

```
      - webserver
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - wordpress:/var/www/html
    command: certonly --webroot --webroot-path=/var/www/html --email sammy@your_domain --
```

This definition tells Compose to pull the `certbot/certbot` image from Docker Hub. It also uses named volumes to share resources with the Nginx container, including the domain certificates and key in `certbot-`

Again, you've used `depends_on` to specify that the `certbot` container should be started once the `webserver` service is running.

You've also included a `command` option that specifies a subcommand to run with the container's default `certbot` command. The `certonly` subcommand will obtain a certificate with the following options:

- `--webroot`: This tells Certbot to use the webroot plugin to place files in the webroot folder for authentication. This plugin depends on the HTTP-01 validation method, which uses an HTTP request to prove that Certbot can access resources from a server that responds to a given domain name.
- `--webroot-path`: This specifies the path of the webroot directory.
- `--email`: Your preferred email for registration and recovery.
- `--agree-tos`: This specifies that you agree to ACME's Subscriber Agreement.
- `--no-eff-email`: This tells Certbot that you do not wish to share your email with the Electronic Frontier Foundation (EFF). Feel free to omit this if you would prefer.
- `--staging`: This tells Certbot that you would like to use Let's Encrypt's staging environment to obtain test certificates. Using this option allows you to test your configuration options and avoid possible domain request limits. For more information about these limits, please read Let's Encrypt's rate limits documentation.
- `-d`: This allows you to specify domain names you would like to apply to your request. In this case, you've included `your_domain` and `www.your_domain`. Be sure to replace these with your own domain.

Below the `certbot` service definition, add your network and volume definitions:

~/wordpress/docker-compose.yml

Copy

```
...
volumes:
  certbot-etc:
  wordpress:
  dbdata:

networks:
```

```
  app-network:
    driver: bridge
```

Your top-level `volumes` key defines the volumes `certbot-etc`, `wordpress`, and `dbdata`. When Docker creates volumes, the contents of the volume are stored in a directory on the host filesystem, `/var/lib/docker/volumes/`, that's managed by Docker. The contents of each volume then get mounted from this directory to any container that uses the volume. In this way, it's possible to share code and data between containers.

The user-defined bridge network `app-network` enables communication between your containers since they are on the same Docker daemon host. This streamlines traffic and communication within the application, as it opens all ports between containers on the same bridge network without exposing any ports to the outside world. Thus, your `db`, `wordpress`, and `webserver` containers can communicate with each other, and you only need to expose port `80` for front-end access to the application.

The following is `docker-compose.yml` file in its entirety:

~/wordpress/docker-compose.yml

Copy

```
version: '3'

services:
  db:
    image: mysql:8.0
    container_name: db
    restart: unless-stopped
    env_file: .env
    environment:
      - MYSQL_DATABASE=wordpress
    volumes:
      - dbdata:/var/lib/mysql
    command: '--default-authentication-plugin=mysql_native_password'
    networks:
      - app-network

  wordpress:
    depends_on:
      - db
    image: wordpress:5.1.1-fpm-alpine
    container_name: wordpress
    restart: unless-stopped
    env_file: .env
    environment:
      - WORDPRESS_DB_HOST=db:3306
      - WORDPRESS_DB_USER=$MYSQL_USER
      - WORDPRESS_DB_PASSWORD=$MYSQL_PASSWORD
      - WORDPRESS_DB_NAME=wordpress
    volumes:
```

```
      - wordpress:/var/www/html
    networks:
      - app-network

  webserver:
    depends_on:
      - wordpress
    image: nginx:1.15.12-alpine
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - wordpress:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
    networks:
      - app-network

  certbot:
    depends_on:
      - webserver
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - wordpress:/var/www/html
    command: certonly --webroot --webroot-path=/var/www/html --email sammy@your_domain --

volumes:
  certbot-etc:
  wordpress:
  dbdata:

networks:
  app-network:
    driver: bridge
```

Save and close the file when you are finished editing.

With your service definitions in place, you are ready to start the containers and test your certificate requests.

Step 4 — Obtaining SSL Certificates and Credentials

Start your containers with the `docker-compose up` command, which will create and run your containers in the order you have specified. By adding the `-d` flag, the command will run the `db`, `wordpress`, and `webserver` containers in the background:

```
docker-compose up -d
```

Copy

The following output confirms that your services have been created:

```
Output
Creating db ... done
Creating wordpress ... done
Creating webserver ... done
Creating certbot   ... done
```

Using `docker-compose ps`, check the status of your services:

```
docker-compose ps
```

Copy

Once complete, your `db`, `wordpress`, and `webserver` services will be `Up` and the `certbot` container will have exited with a `0` status message:

```
Output
  Name                 Command               State           Ports
-------------------------------------------------------------------------
certbot     certbot certonly --webroot ...   Exit 0
db          docker-entrypoint.sh --def ...   Up       3306/tcp, 33060/tcp
webserver   nginx -g daemon off;             Up       0.0.0.0:80->80/tcp
wordpress   docker-entrypoint.sh php-fpm     Up       9000/tcp
```

Anything other than `Up` in the `State` column for the `db`, `wordpress`, or `webserver` services, or an exit status other than `0` for the `certbot` container means that you may need to check the service logs with the `docker-compose logs` command:

```
docker-compose logs service_name
```

Copy

You can now check that your certificates have been mounted to the `webserver` container with `docker-compose exec`:

```
docker-compose exec webserver ls -la /etc/letsencrypt/live
```

Copy

Once your certificate requests succeed, the following is the output:

```
Output
total 16
drwx------    3 root     root          4096 May 10 15:45 .
drwxr-xr-x    9 root     root          4096 May 10 15:45 ..
-rw-r--r--    1 root     root           740 May 10 15:45 README
drwxr-xr-x    2 root     root          4096 May 10 15:45 your_domain
```

Now that you know your request will be successful, you can edit the `certbot` service definition to remove the `--staging` flag.

Open `docker-compose.yml`:

```
nano docker-compose.yml
```

Copy

Find the section of the file with the `certbot` service definition, and replace the `--staging` flag in the `command` option with the `--force-renewal` flag, which will tell Certbot that you want to request a new certificate with the same domains as an existing certificate. The following is the `certbot` service definition with the updated flag:

~/wordpress/docker-compose.yml

Copy

```
...
  certbot:
    depends_on:
      - webserver
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - certbot-var:/var/lib/letsencrypt
      - wordpress:/var/www/html
    command: certonly --webroot --webroot-path=/var/www/html --email sammy@your_domain --
...
```

You can now run `docker-compose up` to recreate the `certbot` container. You will also include the `--no-deps` option to tell Compose that it can skip starting the `webserver` service, since it is already running:

```
docker-compose up --force-recreate --no-deps certbot
```

Copy

The following output indicates that your certificate request was successful:

```
Output
Recreating certbot ... done
Attaching to certbot
certbot      | Saving debug log to /var/log/letsencrypt/letsencrypt.log
certbot      | Plugins selected: Authenticator webroot, Installer None
certbot      | Renewing an existing certificate
certbot      | Performing the following challenges:
certbot      | http-01 challenge for your_domain
certbot      | http-01 challenge for www.your_domain
certbot      | Using the webroot path /var/www/html for all unmatched domains.
certbot      | Waiting for verification...
certbot      | Cleaning up challenges
certbot      | IMPORTANT NOTES:
certbot      |  - Congratulations! Your certificate and chain have been saved at:
certbot      |    /etc/letsencrypt/live/your_domain/fullchain.pem
certbot      |    Your key file has been saved at:
certbot      |    /etc/letsencrypt/live/your_domain/privkey.pem
certbot      |    Your cert will expire on 2019-08-08. To obtain a new or tweaked
certbot      |    version of this certificate in the future, simply run certbot
certbot      |    again. To non-interactively renew *all* of your certificates, run
certbot      |    "certbot renew"
certbot      |  - Your account credentials have been saved in your Certbot
certbot      |    configuration directory at /etc/letsencrypt. You should make a
certbot      |    secure backup of this folder now. This configuration directory will
certbot      |    also contain certificates and private keys obtained by Certbot so
certbot      |    making regular backups of this folder is ideal.
certbot      |  - If you like Certbot, please consider supporting our work by:
certbot      |
certbot      |    Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
certbot      |    Donating to EFF:                    https://eff.org/donate-le
certbot      |
certbot exited with code 0
```

With your certificates in place, you can move on to modifying your Nginx configuration to include SSL.

Step 5 — Modifying the Web Server Configuration and Service Definition

Enabling SSL in your Nginx configuration will involve adding an HTTP redirect to HTTPS, specifying your SSL certificate and key locations, and adding security parameters and headers.

Since you are going to recreate the `webserver` service to include these additions, you can stop it now:

```
docker-compose stop webserver
```

Copy

Before modifying the configuration file, get the recommended Nginx security parameter from Certbot using `curl`:

```
curl -sSLo nginx-conf/options-ssl-nginx.conf https://raw.githubusercontent.com/ce
```
Copy
cer

This command will save these parameters in a file called `options-ssl-nginx.conf`, located in the `nginx-conf` directory.

Next, remove the Nginx configuration file you created earlier:

```
rm nginx-conf/nginx.conf
```
Copy

Create and open another version of the file:

```
nano nginx-conf/nginx.conf
```
Copy

Add the following code to the file to redirect HTTP to HTTPS and to add SSL credentials, protocols, and security headers. Remember to replace `your_domain` with your own domain:

~/wordpress/nginx-conf/nginx.conf

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

        server_name your_domain www.your_domain;

        location ~ /.well-known/acme-challenge {
                allow all;
                root /var/www/html;
        }

        location / {
                rewrite ^ https://$host$request_uri? permanent;
        }
}

server {
        listen 443 ssl http2;
        listen [::]:443 ssl http2;
        server_name your_domain www.your_domain;

        index index.php index.html index.htm;
```

```
        root /var/www/html;

        server_tokens off;

        ssl_certificate /etc/letsencrypt/live/ your_domain /fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/ your_domain /privkey.pem;

        include /etc/nginx/conf.d/options-ssl-nginx.conf;

        add_header X-Frame-Options "SAMEORIGIN" always;
        add_header X-XSS-Protection "1; mode=block" always;
        add_header X-Content-Type-Options "nosniff" always;
        add_header Referrer-Policy "no-referrer-when-downgrade" always;
        add_header Content-Security-Policy "default-src * data: 'unsafe-eval' 'unsafe-inli
        # add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; prelo
        # enable strict transport security only if you understand the implications

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

        location ~ \.php$ {
                try_files $uri =404;
                fastcgi_split_path_info ^(.+\.php)(/.+)$;
                fastcgi_pass wordpress:9000;
                fastcgi_index index.php;
                include fastcgi_params;
                fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
                fastcgi_param PATH_INFO $fastcgi_path_info;
        }

        location ~ /\.ht {
                deny all;
        }

        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;
        }
}
```

The HTTP server block specifies the webroot for Certbot renewal requests to the `.well-known/acme-challenge` directory. It also includes a rewrite directive that directs HTTP requests to the root directory to HTTPS.

The HTTPS server block enables `ssl` and `http2`. To read more about how HTTP/2 iterates on HTTP protocols and the benefits it can have for website performance, please read the introduction to How To Set Up Nginx with HTTP/2 Support on Ubuntu 18.04.

This block also includes your SSL certificate and key locations, along with the recommended Certbot security parameters that you saved to `nginx-conf/options-ssl-nginx.conf`.

Additionally, included are some security headers that will enable you to get **A** ratings on things like the SSL Labs and Security Headers server test sites. These headers include `X-Frame-Options`, `X-Content-Type-Options`, `Referrer Policy`, `Content-Security-Policy`, and `X-XSS-Protection`. The HTTP `Strict Transport Security` (HSTS) header is commented out — enable this only if you understand the implications and have assessed its "preload" functionality.

Your `root` and `index` directives are also located in this block, as are the rest of the WordPress-specific location blocks discussed in Step 1.

Once you have finished editing, save and close the file.

Before recreating the `webserver` service, you will need to add a `443` port mapping to your `webserver` service definition.

Open your `docker-compose.yml` file:

```
nano docker-compose.yml
```

Copy

In the `webserver` service definition, add the following port mapping:

~/wordpress/docker-compose.yml

Copy

```
...
  webserver:
    depends_on:
      - wordpress
    image: nginx:1.15.12-alpine
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - wordpress:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
```

```
    networks:
      - app-network
```

Here is the complete `docker-compose.yml` file after the edits:

~/wordpress/docker-compose.yml

Copy

```
version: '3'

services:
  db:
    image: mysql:8.0
    container_name: db
    restart: unless-stopped
    env_file: .env
    environment:
      - MYSQL_DATABASE=wordpress
    volumes:
      - dbdata:/var/lib/mysql
    command: '--default-authentication-plugin=mysql_native_password'
    networks:
      - app-network

  wordpress:
    depends_on:
      - db
    image: wordpress:5.1.1-fpm-alpine
    container_name: wordpress
    restart: unless-stopped
    env_file: .env
    environment:
      - WORDPRESS_DB_HOST=db:3306
      - WORDPRESS_DB_USER=$MYSQL_USER
      - WORDPRESS_DB_PASSWORD=$MYSQL_PASSWORD
      - WORDPRESS_DB_NAME=wordpress
    volumes:
      - wordpress:/var/www/html
    networks:
      - app-network

  webserver:
    depends_on:
      - wordpress
    image: nginx:1.15.12-alpine
    container_name: webserver
    restart: unless-stopped
    ports:
      - "80:80"
```

```
      - "443:443"
    volumes:
      - wordpress:/var/www/html
      - ./nginx-conf:/etc/nginx/conf.d
      - certbot-etc:/etc/letsencrypt
    networks:
      - app-network

  certbot:
    depends_on:
      - webserver
    image: certbot/certbot
    container_name: certbot
    volumes:
      - certbot-etc:/etc/letsencrypt
      - wordpress:/var/www/html
    command: certonly --webroot --webroot-path=/var/www/html --email sammy@your_domain --

volumes:
  certbot-etc:
  wordpress:
  dbdata:

networks:
  app-network:
    driver: bridge
```

Save and close the file when you are finished editing.

Recreate the `webserver` service:

```
docker-compose up -d --force-recreate --no-deps webserver
```

Copy

Check your services with `docker-compose ps`:

```
docker-compose ps
```

Copy

The output should indicate that your `db`, `wordpress`, and `webserver` services are running:

```
Output
  Name                 Command               State                    Ports
-------------------------------------------------------------------------------------
certbot     certbot certonly --webroot ...   Exit 0
db          docker-entrypoint.sh --def ...   Up       3306/tcp, 33060/tcp
```

```
webserver   nginx -g daemon off;             Up       0.0.0.0:443->443/tcp, 0.0.0.0:80->80
wordpress   docker-entrypoint.sh php-fpm     Up       9000/tcp
```

With your containers running, you can complete your WordPress installation through the web interface.

Step 6 — Completing the Installation Through the Web Interface

With your containers running, finish the installation through the WordPress web interface.

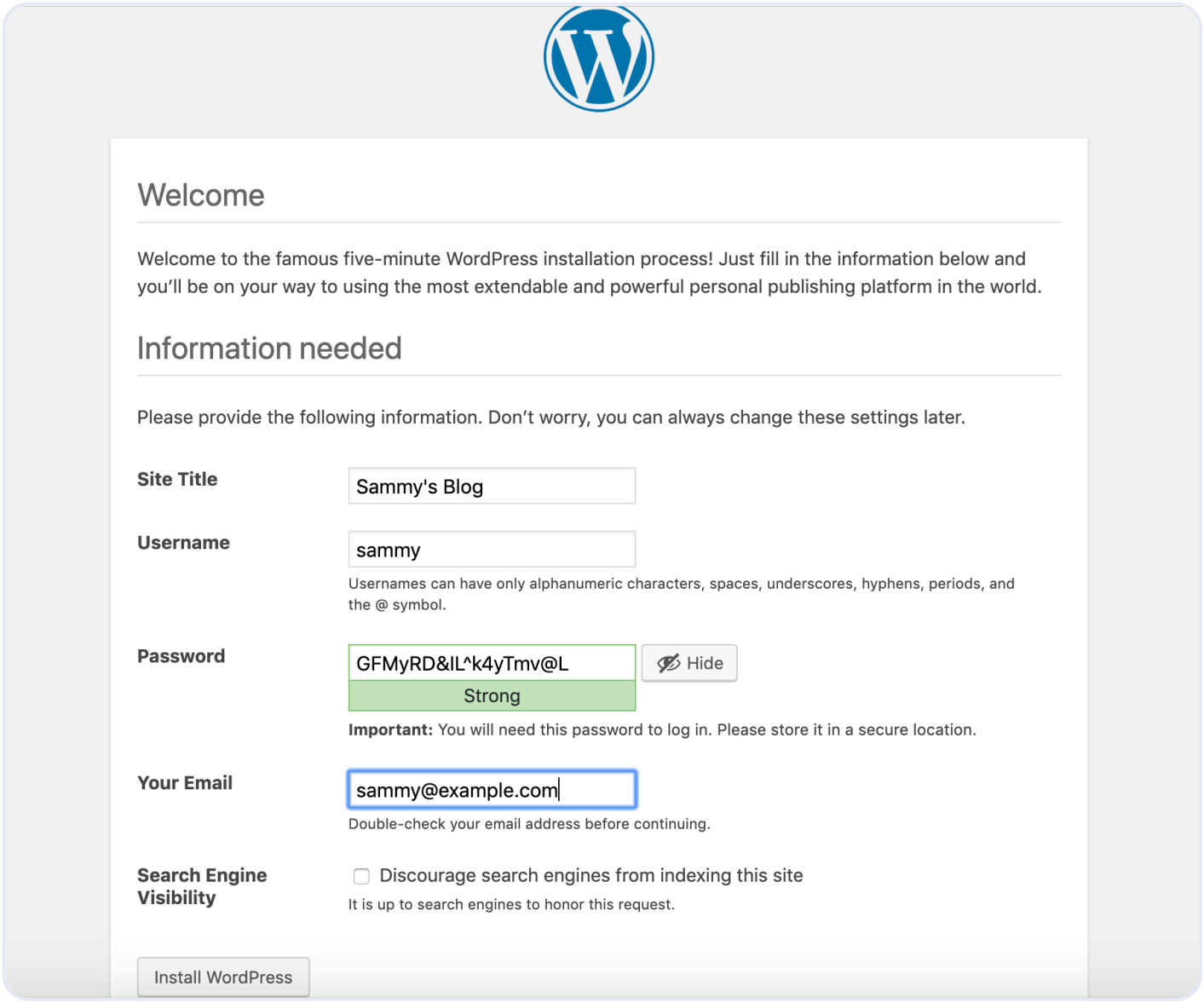
In your web browser, navigate to your server's domain. Remember to substitute `your_domain` with your own domain name:

```
https://your_domain
```

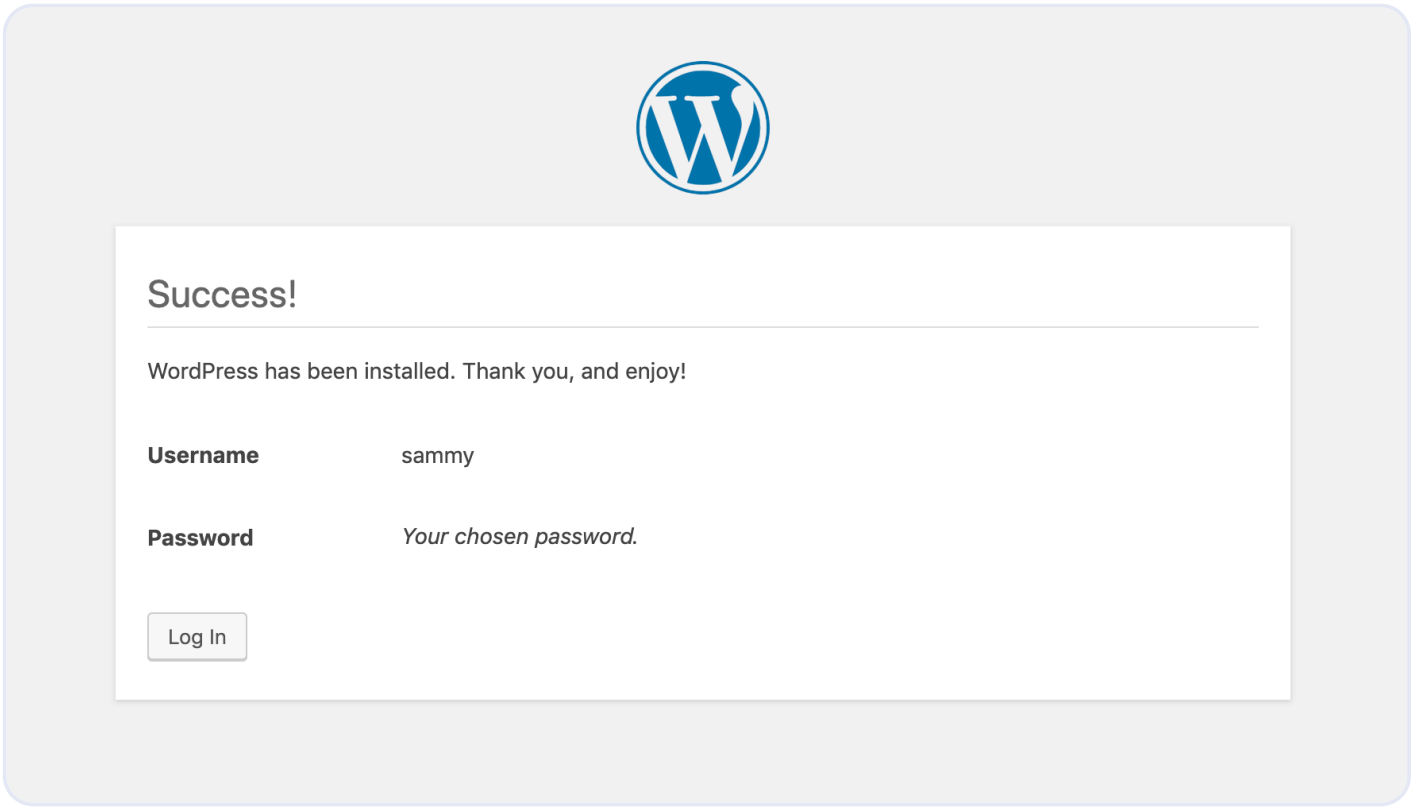
Select the language you would like to use:

After clicking **Continue**, you will land on the main setup page, where you will need to pick a name for your site and a username. It's a good idea to choose a memorable username here (rather than "admin") and a strong password. You can use the password that WordPress generates automatically or create your own.

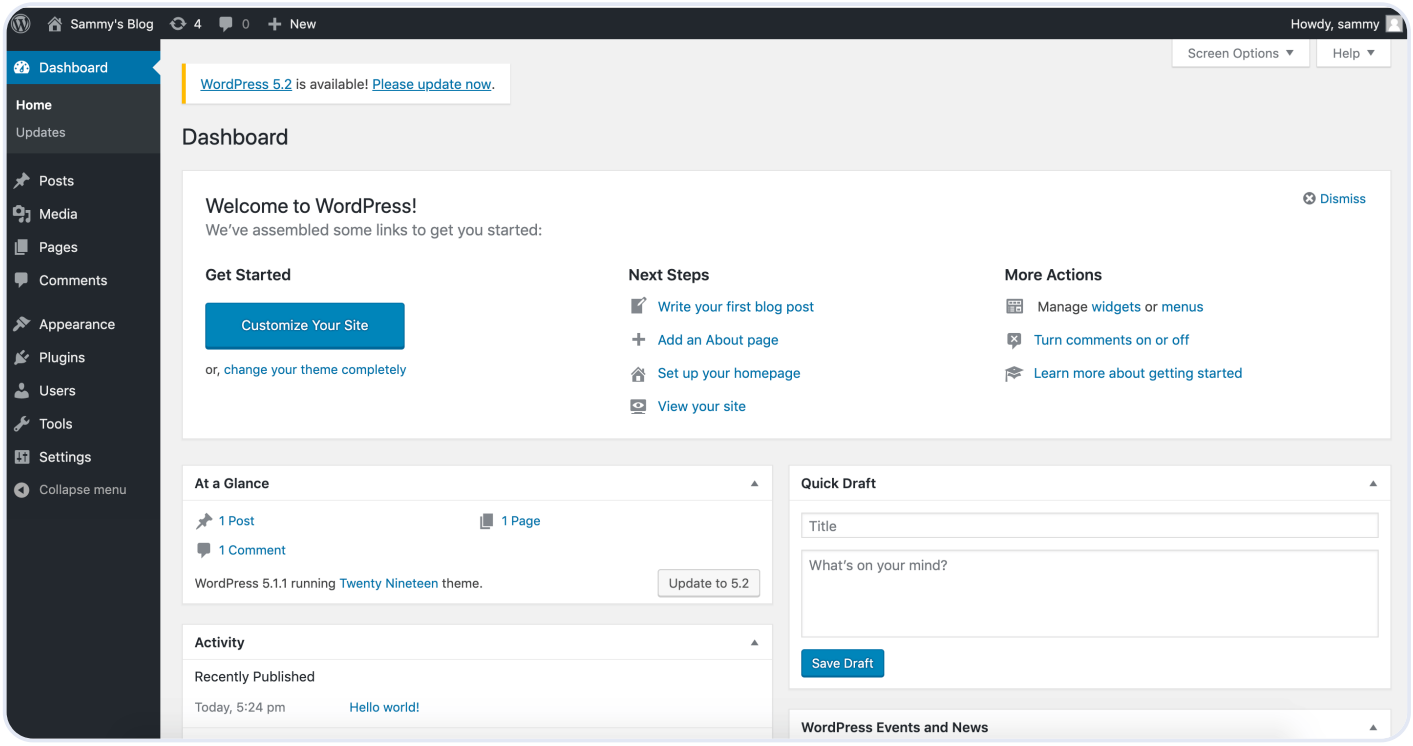
Finally, you will need to enter your email address and decide whether or not you want to discourage search engines from indexing your site:



Clicking on **Install WordPress** at the bottom of the page will take you to a login prompt:



Once logged in, you will have access to the WordPress administration dashboard:



With your WordPress installation complete, you can take steps to ensure that your SSL certificates will renew automatically.

Step 7 — Renewing Certificates

Let's Encrypt certificates are valid for 90 days. You can set up an automated renewal process to ensure that they do not lapse. One way to do this is to create a job with the `cron` scheduling utility. In the following

example, you will create a `cron` job to periodically run a script that will renew your certificates and reload your Nginx configuration.

First, open a script called `ssl_renew.sh`:

```
nano ssl_renew.sh
```

Copy

Add the following code to the script to renew your certificates and reload your web server configuration. Remember to replace the example username here with your own non-**root** username:

~/wordpress/ssl_renew.sh

```
#!/bin/bash

COMPOSE="/usr/local/bin/docker-compose --no-ansi"
DOCKER="/usr/bin/docker"

cd /home/sammy/wordpress/
$COMPOSE run certbot renew --dry-run && $COMPOSE kill -s SIGHUP webserver
$DOCKER system prune -af
```

Copy

This script first assigns the `docker-compose` binary to a variable called `COMPOSE`, and specifies the `--no-ansi` option, which will run `docker-compose` commands without ANSI control characters. It then does the same with the `docker` binary. Finally, it changes to the `~/wordpress` project directory and runs the following `docker-compose` commands:

- `docker-compose run`: This will start a `certbot` container and override the `command` provided in your `certbot` service definition. Instead of using the `certonly` subcommand, the `renew` subcommand is used, which will renew certificates that are close to expiring. Also included is the `--dry-run` option to test your script.
- `docker-compose kill`: This will send a `SIGHUP` signal to the `webserver` container to reload the Nginx configuration.

It then runs `docker system prune` to remove all unused containers and images.

Close the file when you are finished editing. Make it executable with the following command:

```
chmod +x ssl_renew.sh
```

Copy

Next, open your **root** `crontab` file to run the renewal script at a specified interval:

```
sudo crontab -e
```

Copy

If this is your first time editing this file, you will be asked to choose an editor:

```
Output
no crontab for root - using an empty one

Select an editor.  To change later, run 'select-editor'.
  1. /bin/nano        <---- easiest
  2. /usr/bin/vim.basic
  3. /usr/bin/vim.tiny
  4. /bin/ed

Choose 1-4 [1]:
...
```

At the very bottom of this file, add the following line:

crontab

```
...
*/5 * * * * /home/ sammy /wordpress/ssl_renew.sh >> /var/log/cron.log 2>&1
```

This will set the job interval to every five minutes, so you can test whether or not your renewal request has worked as intended. A log file, `cron.log`, is created to record relevant output from the job.

After five minutes, check `cron.log` to confirm whether or not the renewal request has succeeded:

```
tail -f /var/log/cron.log
```

Copy

The following output confirms a successful renewal:

```
Output
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
** DRY RUN: simulating 'certbot renew' close to cert expiry
**          (The test certificates below have not been saved.)

Congratulations, all renewals succeeded. The following certs have been renewed:
  /etc/letsencrypt/live/ your_domain /fullchain.pem (success)
** DRY RUN: simulating 'certbot renew' close to cert expiry
**          (The test certificates above have not been saved.)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Exit out by entering `CTRL+C` in your terminal.

You can modify the `crontab` file to set a daily interval. To run the script every day at noon, for example, you would modify the last line of the file like the following:

crontab

```
...
0 12 * * * /home/ sammy /wordpress/ssl_renew.sh >> /var/log/cron.log 2>&1
```

You will also want to remove the `--dry-run` option from your `ssl_renew.sh` script:

~/wordpress/ssl_renew.sh

```
#!/bin/bash

COMPOSE="/usr/local/bin/docker-compose --no-ansi"
DOCKER="/usr/bin/docker"

cd /home/ sammy /wordpress/
$COMPOSE run certbot renew && $COMPOSE kill -s SIGHUP webserver
$DOCKER system prune -af
```

Copy

Your `cron` job will ensure that your Let's Encrypt certificates don't lapse by renewing them when they are eligible. You can also set up log rotation with the Logrotate utility on Ubuntu 22.04 / 20.04 to rotate and compress your log files.

Conclusion

In this tutorial, you used Docker Compose to create a WordPress installation with an Nginx web server. As part of this workflow, you obtained TLS/SSL certificates for the domain you want associated with your WordPress site. Additionally, you created a `cron` job to renew these certificates when necessary.

As additional steps to improve site performance and redundancy, you can consult the following articles on delivering and backing up WordPress assets:

- How to Speed Up WordPress Asset Delivery Using DigitalOcean Spaces CDN.
- How To Back Up a WordPress Site to Spaces.
- How To Store WordPress Assets on DigitalOcean Spaces.

If you are interested in exploring a containerized workflow with Kubernetes, you can also check out How To Set Up WordPress with MySQL on Kubernetes Using Helm.