# Modern CI/CD for Static Websites: Automating Deployment with Docker, Nginx, and Shell Scripts



## Table of Contents

---

## Purpose

This project is a step-by-step tutorial designed to simulate how CI/CD pipelines are built and scaled for real-world applications. Starting from the basics, you'll learn how to set up a robust CI/CD pipeline for static websites using Docker, Nginx, and Shell scripts. By following each step, you'll understand how to scale from a simple setup to a fully functional CI/CD pipeline ready for production.

This setup is reusable for any static website, whether it’s:

- A personal portfolio website.
- A website to represent your services.
- A business website needing regular updates.

By the end of this tutorial, you'll have a foundational CI/CD pipeline that shifts your focus from deployment headaches to content creation and design updates. You’ll no longer worry about how to:

- Serve your website.
- Set up a custom domain.
- Integrate SSL certificates.

Instead, your focus will shift entirely to creating and updating the content and beautifying your website with HTML, CSS, and JavaScript.

---

## How to Use This Repository and Study

This tutorial is designed as a **step-by-step guide**, with each step building upon the previous one. To get the most out of this project, follow these instructions:

### 1. Follow Each Step Sequentially

- Each step is carefully structured to simulate the process of building and scaling a CI/CD pipeline in real-world applications.
- **Do not skip steps**, as every new step builds on the work from the previous one.

### 2. Instructions for Each Step

- In each step, the `README.md` will provide:
  - **Tasks to Complete**: Specific actions you need to perform (e.g., creating files, writing code).
  - **Tests**: A way to verify that you have completed the step correctly.

### 3. Encourage Self-Learning

- If you encounter challenges during a step:
  - **Google it**: Use search engines to find solutions.
  - **Use ChatGPT**: Ask for guidance or clarification.
  - **Visit Stack Overflow**: Research common issues and solutions.
- Spending time troubleshooting on your own is an invaluable part of the learning process.

### 4. Get Support When Needed

If you're still unable to resolve an issue after self-search:

- **Contact Us**: Send a support request through our app for assistance.
- **Book a Mentorship Session**: For more in-depth help, you can schedule a mentorship session with us via the app.
- For more information about these services, feel free to reach us at **info@iswad.tech**.

### 5. Use the Branches for Guidance

- Each step has a corresponding branch named `step-{STEP_NUMBER}` (e.g., `step-01`, `step-02`):
  - Switch to the relevant branch to see how the step should be structured:

    ```
    git checkout step-{STEP_NUMBER}
    ```

  - Review the comments in the code, where every line or important section is explained.
- If you want to compare your solution with mine, you can pull the code from the branch:

  ```
  git pull origin step-{STEP_NUMBER}
  ```

### 6. Compare Changes Between Steps

- To fully understand what changed and why, use Git to compare files and folders updated in the current step:
  - **Using Pull Requests**: If you're using a repository with pull requests for each step, review the diff to analyze updates.
  - **Using Command Line**:
    - Compare changes between steps:

      ```
      ```

```
git diff step-{PREVIOUS_STEP_NUMBER} step-
{CURRENT_STEP_NUMBER}
```

        - This will show the differences between the two steps, helping you understand the updates made.

### 7. Track Your Progress

After completing each step:

- **Mark the Step as Completed**: Update the status of that step in the app to indicate that it has been successfully completed.
- **Share Optional Feedback**:
    - Include what you learned during the step or what challenges you faced and how you solved them.
    - This information will help:
        - Keep track of your progress and make you more committed to your work.
        - Assess whether you have successfully completed the step.
        - Enable future AI-powered features (currently under development) that will:
            - Help you generate a better resume.
            - Suggest relevant topics and personalized roadmaps to achieve your goal of becoming a professional or senior developer.

---

By following this structured process, you'll gain both practical knowledge and confidence in building CI/CD pipelines for static websites. Additionally, our tools and features will help you stay motivated, track progress, and work towards your long-term career goals.

---

## Who Is This For?

This project is for anyone who wants to learn how to automate and scale website deployments, including:

- **DevOps Engineers**: Build expertise in automating pipelines for static websites.
- **Full-Stack Engineers**: Level up by learning the basics of DevOps.
- **Front-End Developers**: Learn how to publish your designs professionally.
- **UI/UX Designers**: If you can export HTML, CSS, and JavaScript or have basic knowledge of it, this guide will show you how to publish your app.
- **Professionals and Service Providers**: Build and scale your website for your services or business.

---

## What You'll Learn

By completing this tutorial, you'll gain:

- The ability to create and manage **Dockerfiles** for static websites.
- Hands-on experience configuring **Nginx** for efficient static file serving.
- Skills to write **Shell scripts** for automating CI/CD pipelines.

- An understanding of CI/CD concepts and how to apply them to real-world projects.
- A reusable deployment pipeline that works for any static website.

---

## Prerequisites

Before you begin, ensure you have the following installed:

- **Git**: Install Git.
- **GitHub account**: For managing the repository and hosting code.
- **Docker**: Install Docker.
- **Shell**: A Unix-based shell (bash/zsh) for running scripts.
- **Basic Knowledge**: Familiarity with HTML, CSS, and JavaScript.

---

## Step 01: Preparing the Development Configuration

### Goal of This Step

In this step, you will:

1. Set up the development environment for your static website.
2. Learn the basics of **Nginx**, **Docker**, and **Docker Compose** to containerize and serve the app.
3. Run the app locally using `docker-compose` and verify that everything is working as expected.

By the end of this step, you'll have a minimal CI/CD setup running locally, serving your static website.

---

### Concepts to Learn

#### 1. Nginx: The Backbone of Modern Web Servers

**Theoretical Overview**: Nginx is a web server used to efficiently serve files (like HTML, CSS, and JavaScript) or act as a reverse proxy. It's widely used in real-world scenarios for its performance and ability to handle thousands of simultaneous requests.

**Practical Analogy**: Think of Nginx as a **waiter** in a restaurant:

- When a user (customer) makes a request, like "Show me the homepage," Nginx fetches the `index.html` file (menu) and delivers it to the user.
- If the customer requests additional resources (like CSS or images), Nginx fetches and serves them quickly.

**Real-World Example**:

- Websites like **Netflix**, **YouTube**, and **Amazon** use Nginx to serve static files like videos, images, and stylesheets.

**How Nginx Works in This Project**:

- It will:
    1. Serve your `index.html` file to users visiting the site.

2. Look for additional static files (like CSS and JavaScript) and deliver them to the browser.

---

**2. Docker: Ensuring Consistency**

**Theoretical Overview**: Docker packages your app and its environment into a container, ensuring it runs the same way across all machines, from your laptop to a production server.

**Practical Analogy**: Think of Docker as a **portable kitchen** that includes:

- The ingredients (your app files).
- The tools (dependencies like Nginx).
- The oven (a runtime environment).

**Real-World Example**:

- Companies like **Spotify** and **Google** use Docker to make sure their apps run identically across development and production environments.

**How Docker Works in This Project**:

- It will:
  1. Package your app files and Nginx into a container.
  2. Serve your app consistently, no matter where it's running.

---

**3. Docker Compose: Managing Multiple Services**

**Theoretical Overview**: Docker Compose lets you define and manage multiple services in one file (`docker-compose.yml`) and run them together with a single command.

**Practical Analogy**: Think of Docker Compose as a **kitchen manager** that coordinates multiple stations:

- One station serves files (Nginx).
- Another could handle data storage (a database).

**Real-World Example**:

- An e-commerce platform might use Docker Compose to manage:
  - A front-end service (React or Angular).
  - A back-end API (Node.js or Django).
  - A database (PostgreSQL or MongoDB).

**How Docker Compose Works in This Project**:

- It will:
  1. Define the Nginx service.
  2. Map your local files to the container for live updates.
  3. Expose port `80` so you can access the app in your browser.

---

## Practical Steps

### 1. Create the Folder Structure

- At the root of your repository:
  - Create a folder named `site` to store all your website files (HTML, CSS, JavaScript, images).
    - Add a file named `index.html` with minimal content (e.g., "Hello, world!").
    - Add a file named `404.html` to serve as a custom 404 error page.
  - Create a folder named `nginx` to store Nginx configuration files.
    - Add a file named `default-dev.conf` for your Nginx configuration.
    - Add a file named `Dockerfile.dev` for building the Nginx Docker image.
  - Create a file named `docker-compose-dev.yml` in the root directory for your Docker Compose setup.

---

### 2. Write the Nginx Configuration

- Inside the `nginx/default-dev.conf` file, write a basic configuration to:
  - Serve files from `/var/www/app`.
  - Look for `index.html` as the default file when users visit the root of your site.
  - Include a fallback to a `404.html` file for missing resources.

```
server {
    # The server listens for incoming HTTP requests on port 80 (default
for HTTP).
    listen 80;

    # Defines the root directory where the website files are stored and
served from.
    root /var/www/app;

    # Specifies the default file to serve when a user visits the root URL.
    index index.html;

    # Defines a custom 404 error page to display when a file is not found.
    error_page 404 /404.html;


    # Handles all other requests to the root or subpaths.
    location / {
        # Attempts to serve the requested file or path in this order:
        # 1. An exact file match (e.g., /about.html).
        # 2. A file with .html appended (e.g., /about -> /about.html).
        # 3. A directory with the name (e.g., /about/).
        # 4. If none of the above exist, serves the custom 404 error page.
        try_files $uri $uri.html $uri/ /404.html;
    }
}
```

For building nginx image with the appropriate configuration, inside Dockerfile.dev file add:

```
# This Dockerfile sets up an Nginx container for serving static files in a
development environment.
# - It uses a lightweight Alpine-based Nginx image.
# - Copies a custom Nginx configuration to the container.
# - Prepares a directory for static files with appropriate permissions.

# Use a lightweight Nginx image based on Alpine Linux for efficiency.
FROM nginx:1.20.2-alpine

# Copy the custom Nginx configuration file # to the container's default
configuration location.
COPY ./default-dev.conf /etc/nginx/conf.d/default.conf

# Create a directory for static files within the container.
# Set permissions to allow read, write, and execute for the owner, and
read/execute for others.
RUN mkdir -p /var/www/app/static && \
    chmod 755 /var/www/app/static
```

---

### 3. Write the Docker Compose File

- In the root directory, define the `docker-compose-dev.yml` file to:
  - Set up Nginx as a service.
  - Map your local `site` folder to `/var/www/app` inside the container.
  - Map the `nginx` folder for custom configurations.
  - Expose port `80` for local access.

```
# This Docker Compose file defines a service for running an Nginx
container in development mode.
# It builds a custom Docker image for Nginx using a Dockerfile located in
the "nginx" folder.
# The "site" folder on the host is mapped into the container to serve
static website files.
# Port 80 on the host is mapped to port 80 in the container, allowing
access via http://localhost.

services:
  nginx: # Defines a service named "nginx"
    restart: always # Ensures the container restarts automatically if it
stops or crashes
    build: # Configuration for building the Docker image
      context: ./nginx # Specifies the build context (folder containing
Dockerfile and related files)
      dockerfile: Dockerfile.dev # Specifies the Dockerfile to use for
building the image
    ports:
      - "80:80" # Maps port 80 on the host to port 80 in the container
(access via http://localhost)
    volumes:
```

```
      - ./site:/var/www/app # Maps the "site" folder on the host to
"/var/www/app" inside the container
```

---

#### 4. Folder Structure

At the end of this step, your folder structure should look like this:

```
.
├── nginx/
│   ├── default-dev.conf   # Write your Nginx configuration in this file
│   └── Dockerfile.dev     # Dockerfile to build the Nginx image
├── site/
│   ├── index.html         # Write your minimal HTML file here (e.g.,
"Hello, world!" app)
│   └── 404.html           # Custom 404 error page for missing resources
├── docker-compose-dev.yml # Define your Docker Compose setup for
development
```

#### 5. Run the App

- Open a terminal or Bash at the root directory where `docker-compose-dev.yml` is located.
- Run the following command to start the app:

```
docker-compose -f docker-compose-dev.yml up --build -d
```

### 6. Verify Everything is Working

1. Visit http://localhost in your browser. You should see the content of your `index.html` file.
2. Modify the `index.html` file to display a “Hello, world!” message.
3. Refresh the browser to ensure the changes are reflected.

---

### 6. Test Your Setup

After completing this step, verify the following:

- Does visiting http://localhost display the content of your `index.html` file?
- Are changes to the `index.html` file reflected in the browser after refreshing?

---

# Step 02: Serving Static Files and Improving Performance with Gzip

## Goal of This Step

In this step, you will:

1. Update your Nginx configuration to serve static files (e.g., CSS, JS, images) from a dedicated `static` folder.
2. Optimize your website's performance by enabling Gzip compression.
3. Test the setup by adding CSS and JavaScript files to the `static` folder and verifying that they are served correctly.

By the end of this step, you'll have a more functional setup capable of serving all your static assets efficiently with improved performance.

---

## Concepts to Learn

### 1. Static File Serving

Nginx can efficiently serve static files like CSS, JS, and images directly from a specific folder. This avoids unnecessary processing and delivers these files to the user quickly.

- **Practical Example**:
  - Websites like **Amazon** and **Netflix** use Nginx to serve static assets (e.g., stylesheets, images, and JavaScript) from dedicated directories.
- **How It Works Here**:
  - We'll configure Nginx to map `/static/` URLs to the `static` folder inside your project directory (`/var/www/app/static`).

### 2. Gzip Compression

Gzip reduces the size of text-based files (like HTML, CSS, and JavaScript) before sending them to the browser, resulting in faster load times.

- **Practical Example**:
  - Gzip is widely used by high-traffic sites to save bandwidth and improve performance.
- **How It Works Here**:
  - We'll enable Gzip in Nginx and configure it to compress supported file types like `text/css`, `application/javascript`, and `text/html`.

---

## Practical Steps

### 1. Add Static File Support in Nginx

1. Update your `nginx/default-dev.conf` file to include the following blocks:

```
location /static/ {
     # Maps requests for /static/ to the static folder in your
project.
     alias /var/www/app/static;
 }
```

```
    location ~* \.(gif|jpe?g|png|svg)$ {
        # Caches these file types for as long as possible.
        expires max;
        # Ensures long-term caching.
        add_header Cache-Control "public, max-age=31536000, immutable";
    }


    location ~* \.(css|js|json)$ {
        # Ensures proper caching behavior.
        add_header Cache-Control "no-cache, must-revalidate";
    }
```

### 2. Add Gzip Configuration

- Create a new file in the `nginx` folder named `gzip.conf`.
- Add the following lines to enable Gzip:

```
gzip on;
gzip_types text/plain text/css application/json application/javascript
text/xml application/xml application/xml+rss text/javascript;
gzip_vary on;
gzip_proxied any;
# Compression level: balance between performance and CPU usage.
gzip_comp_level 6;
gzip_buffers 16 8k;
gzip_http_version 1.1;
# Only compress files larger than 256 bytes.
gzip_min_length 256;
```

### Update Your Dockerfile.dev to Include the Gzip Configuration

Add the following line to your `Dockerfile.dev`:

```
# Copies the Gzip configuration into the container.
COPY ./gzip.conf /etc/nginx/conf.d/gzip.conf
```

### 3. Add Static Files

1. Inside the `site` folder, create a `static` directory.
2. Inside the static folder, create two subfolders css, js
3. Add some files to test:
    - Inside css folder, create a file named `styles.css` with the following content:

```
body {
  background-color: blue;
  font-family: Arial, sans-serif;
  color: wheat;
}
```

- Inside js folder, create a file named `main.js` with the following content:

```
console.log("Hello, world!");
```

4. Link these files in your `index.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Hello World!</title>
    <link rel="stylesheet" href="/static/css/styles.css" />
  </head>
  <body>
    Hello World!!!

    <script src="/static/js/main.js"></script>
  </body>
</html>
```

### 4. Folder Structure

After completing this step, your folder structure should look like this:

```
.
├── nginx/
│   ├── default-dev.conf   # Updated Nginx configuration for static files
│   ├── gzip.conf          # Gzip compression configuration
│   └── Dockerfile.dev     # Dockerfile for the Nginx service
├── site/
│   ├── index.html         # Updated HTML linking static files
│   └── static/
│       ├── css/
│       │   └── styles.css # Test CSS file
│       ├── js/
│       │   └── main.js    # Test JavaScript file
├── docker-compose-dev.yml # Docker Compose file for development
```

### 5. Run and Test the Setup

1. Restart your Docker containers to apply the changes:

```
docker-compose -f docker-compose-dev.yml up --build -d
```

Open your browser and visit:

- http://localhost: You should see your updated `index.html` with the applied CSS styles.
- Check the browser's developer tools (**Console tab**) to confirm that the JavaScript file is executed.

Verify Gzip compression:

- Use an online tool like Gzip Checker or your browser's **Network tab** to confirm that files like `styles.css` and `scripts.js` are compressed.

### 6. Test Your Setup

- Does visiting http://localhost show the updated page with CSS and JavaScript applied?
- Are static files served from the `/static/` folder?
- Is Gzip compression applied to your text-based files (CSS, JS, HTML)?

By completing this step, you've set up a static file server with Nginx and optimized it with Gzip compression for faster delivery. This serves as the foundation for efficient static asset management in your CI/CD pipeline.

---

# Step 03: Setting Up the Production Environment

## Goal of This Step

In this step, we will:

1. Set up a production-ready environment with a real domain.
2. Configure a Linux Ubuntu server to host the application.
3. Generate SSL certificates for the domain using Certbot to enable HTTPS.
4. Build and deploy a minimal Nginx-based setup for SSL creation.

By the end of this step, you'll have the foundation ready to run your app on a production server with a valid SSL certificate.

---

## Prerequisites

Before proceeding:

1. **Purchase a Domain**:
    - Use providers like **GoDaddy** or others to buy a domain.
2. **Set Up a Linux Server**:

- Purchase a Linux server (preferably Ubuntu) from a provider like **DigitalOcean**.
- DigitalOcean is recommended for its simplicity, features, and pricing.

3. **DNS Configuration**:
   - Create an **A record** in your domain's DNS settings pointing to your server's public IP address.

---

# Steps to Configure the Production Environment

## 1. Create the Nginx Configuration for SSL Creation

1. In the `nginx` folder, create a file named `default-create-ssl.conf` with the following content:

```
server {
 # Configures the server to listen on port 80 for HTTP requests.
 listen 80;

 # Replace `APP_URL` with your actual domain name (e.g., example.com).
 # This specifies the domain that this Nginx configuration will serve.
 server_name APP_URL www.APP_URL;

 # Disables the Nginx version number in error pages and headers for
security.
 server_tokens off;

 location /.well-known/acme-challenge/ {
     # Allows unrestricted access to the `.well-known/acme-challenge`
directory,
     # which is required for Certbot to verify domain ownership.
     allow all;

     # Specifies the directory where Certbot will store temporary
challenge files
     # used to verify the domain when generating an SSL certificate.
     root /var/www/certbot;
 }

 location / {
     # Serves static HTML files from the `/usr/share/nginx/html`
directory.
     root /usr/share/nginx/html;

     # Specifies the default files (index.html or index.htm) to serve
     # when a user visits the root URL.
     index index.html index.htm;

     # Attempts to serve the requested file (`$uri`).
     # If the file is not found, it falls back to `/index.html`.
     try_files $uri $uri/ /index.html;
 }
}
```

## 2. Create a Dockerfile for SSL Setup

1. Navigate to the `nginx` folder in your project directory.
2. Create a file named `Dockerfile.create-ssl` and add the following content:

```
# Use the lightweight Alpine-based Nginx image as the base image.
FROM nginx:1.20.2-alpine

# Copy the minimal Nginx configuration for SSL creation into the
container.
COPY ./default-create-ssl.conf /etc/nginx/conf.d/default.conf
```

## 3. Create the Docker Compose File

1. Navigate to the root folder of your project.
2. Create a file named `docker-compose-create-ssl.yml` and add the following content:

```
services:
  nginx:
    restart: always
    # Ensures the Nginx container restarts automatically if it stops
or crashes.

    build:
      context: ./nginx
      # Specifies the build context (the directory containing the
Dockerfile).
      dockerfile: Dockerfile.create-ssl
      # Defines the Dockerfile to use for building the Nginx image.

    ports:
      - "80:80"
      # Maps port 80 on the server to port 80 in the container,
allowing HTTP traffic.

    volumes:
      - ./site:/usr/share/nginx/html
      # Maps the local `site` folder to `/usr/share/nginx/html` in the
container.
      # This folder contains the static files (e.g., index.html) for
the Nginx server.

      - ./nginx/certbot/www:/var/www/certbot
      # Maps the local folder for Certbot's `.well-known` directory to
`/var/www/certbot` in the container.
      # Certbot uses this directory to store temporary challenge files
for domain verification.
```

```
      - ./nginx/certbot/conf:/etc/letsencrypt
      # Maps the local folder for Certbot's configuration and SSL
certificates to `/etc/letsencrypt` in the container.
      # This is where Certbot stores the generated SSL certificates.

  certbot:
    image: certbot/certbot:latest
    # Uses the latest Certbot image from Docker Hub to manage SSL
certificates.

    container_name: certbot
    # Sets the name of the Certbot container for easier
identification.

    volumes:
      - ./nginx/certbot/www:/var/www/certbot
      # Shares the `.well-known` folder with the Certbot container for
domain verification.

      - ./nginx/certbot/conf:/etc/letsencrypt
      # Shares the configuration and SSL certificate folder with the
Certbot container.

    entrypoint: "/bin/sh -c 'trap exit TERM; while :; do sleep 6h &
wait $${!}; certbot renew; done'"
    # Defines the custom entrypoint for the Certbot container:
    # - Keeps the container running in a loop.
    # - Attempts to renew SSL certificates every 6 hours.
    # - Ensures the process exits gracefully if the container stops.
```

### 4. Create the `init-letsencrypt.sample.sh` Script

1. In the root folder of your project, create a file named `init-letsencrypt.sample.sh`.
2. Make the script executable:

```
chmod +x init-letsencrypt.sh
```

3. Add the following code to the file:

```
#!/bin/bash

if ! [ -x "$(command -v docker-compose)" ]; then
echo 'Error: docker-compose is not installed.' >&2
exit 1
fi

# Replace APP_URL with your actual domain (e.g., example.com)
domains=(APP_URL www.APP_URL)
```

```
rsa_key_size=4096
data_path="./nginx/certbot"

# Update EMAIL for Let's Encrypt notifications
email="EMAIL"

staging=0


if [ ! -e "$data_path/conf/options-ssl-nginx.conf" ] || [ ! -e
"$data_path/conf/ssl-dhparams.pem" ]; then
echo "### Downloading recommended TLS parameters ..."
mkdir -p "$data_path/conf"
curl -s https://raw.githubusercontent.com/certbot/certbot/master/certbot-
nginx/certbot_nginx/_internal/tls_configs/options-ssl-nginx.conf >
"$data_path/conf/options-ssl-nginx.conf"
curl -s
https://raw.githubusercontent.com/certbot/certbot/master/certbot/certbot/s
sl-dhparams.pem > "$data_path/conf/ssl-dhparams.pem"
echo
fi

echo "### Creating dummy certificate for $domains ..."
path="/etc/letsencrypt/live/$domains"
mkdir -p "$data_path/conf/live/$domains"
docker-compose -f ./docker-compose-create-ssl.yml run --rm --entrypoint "\
openssl req -x509 -nodes -newkey rsa:$rsa_key_size -days 1\
  -keyout '$path/privkey.pem' \
  -out '$path/fullchain.pem' \
  -subj '/CN=localhost'" certbot
echo


echo "### Starting nginx ..."
docker-compose -f ./docker-compose-create-ssl.yml up --force-recreate -d
nginx
echo

echo "### Deleting dummy certificate for $domains ..."
docker-compose -f ./docker-compose-create-ssl.yml run --rm --entrypoint "\
rm -Rf /etc/letsencrypt/live/$domains && \
rm -Rf /etc/letsencrypt/archive/$domains && \
rm -Rf /etc/letsencrypt/renewal/$domains.conf" certbot
echo


echo "### Requesting Let's Encrypt certificate for $domains ..."
#Join $domains to -d args
domain_args=""
for domain in "${domains[@]}"; do
domain_args="$domain_args -d $domain"
done
```

```
# Select appropriate email arg
case "$email" in
"") email_arg="--register-unsafely-without-email" ;;
*) email_arg="--email $email" ;;
esac

# Enable staging mode if needed
if [ $staging != "0" ]; then staging_arg="--staging"; fi

docker-compose -f ./docker-compose-create-ssl.yml run --rm --entrypoint "\
certbot certonly --webroot -w /var/www/certbot \
  $staging_arg \
  $email_arg \
  $domain_args \
  --rsa-key-size $rsa_key_size \
  --agree-tos \
  --force-renewal" certbot
echo

echo "### Reloading nginx ..."
docker-compose -f ./docker-compose-create-ssl.yml exec nginx nginx -s
reload
```

## 5. SSH into the Server

1. Use the following command to log in to your server:

   ```
   ssh root@IP_ADDRESS
   ```

Replace `IP_ADDRESS` with the public IP address of your server.
This command connects you to your server as the root user.

Once logged in:

## 6. Set Up Git Configuration

**Configure your Git to pull from your private repository using SSH:**

1. **Generate an SSH key on your server**: Go to .ssh folder on the serve:

```
cd .ssh
```

```
ssh-keygen -t rsa -b 4096 -C "OPTINAL COMMENT"
```

This command creates a 4096-bit RSA key pair with a comment containing your email address.

2. **Copy the public key**:

```
cat ~/.ssh/id_rsa.pub
```

Use this command to display the public portion of your SSH key.

3. **Add the key to your GitHub account**:

- Go to **GitHub > Settings > SSH and GPG keys**.
- Click **New SSH Key**, paste the public key, and save it.

**Test the SSH connection**:

```
ssh -T git@github.com
```

If successful, you should see a message like:

```
Hi username! You've successfully authenticated.
```

This confirms that your server can securely connect to GitHub via SSH.

## 7. Deploy the App and Create SSL

1. **Add `init-letsencrypt.sh` to your `.gitignore` file**:
    - Open the `.gitignore` file in the root folder of your project in your local system.
    - Add the following line:

      ```
      init-letsencrypt.sh
      nginx/certbot
      ```

    - This ensures that the sensitive files/folders are not pushed to the GitHub repository.
2. **Push Your Changes to GitHub**:
    - Commit and push the updates:

      ```
      git add .
      git commit -m "Add SSL setup and ignore init-letsencrypt.sh"
      git push origin main
      ```

3. **Pull the Changes on the Server**:

- First, ensure **Git** is installed on the server. If it's not installed, use the following command:

```
apt update
apt install -y git
```

- Navigate to the `/var/www/app` directory on your server:

```
cd /var/www/app
```

- Add your GitHub repository as the `origin` using SSH:

```
git init
git remote add origin git@github.com:yourusername/your-repo-
name.git
```

  - Replace `yourusername/your-repo-name` with the actual username and repository name of your GitHub project.

- Pull the latest changes from your GitHub repository:

```
# master can be replaced with the actual branch name in your
repository
git pull origin master
```

4. **Copy `init-letsencrypt.sh.sample` to `init-letsencrypt.sh`**:

   - Create the working script from the sample file and make it executable:

```
cp init-letsencrypt.sample.sh init-letsencrypt.sh
chmod +x init-letsencrypt.sh
```

     - Update `APP_URL` and `EMAIL` in the copied file.

5. **Install Docker and Docker Compose on the Server**:

   - If Docker and Docker Compose are not already installed, use the following commands to install them:

     **Install Docker**:

```
apt update
apt install -y docker.io
```

```
systemctl start docker
systemctl enable docker
```

**Install Docker Compose**:

```
apt install -y curl
curl -L
"https://github.com/docker/compose/releases/latest/download/docke
r-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
compose
chmod +x /usr/local/bin/docker-compose
```

- Verify that Docker and Docker Compose are installed correctly:

```
docker --version
docker-compose --version
```

6. **Run the `init-letsencrypt.sh` Script**:
   - Execute the script to create and configure SSL certificates:

```
./init-letsencrypt.sh
```

## 8. Verify SSL Creation

Once the script completes successfully, you will see an appropriate success message in the terminal, and you are ready to move to the next step.

## 8. Folder and File Structure

At the end of this step, your project should have the following folder and file structure:

```
.
├── nginx/                          # Contains Nginx configurations and
Dockerfiles
│   ├── default-create-ssl.conf     # Minimal Nginx configuration for
creating SSL certificates
│   ├── default-dev.conf            # Nginx configuration for the
development environment
│   ├── gzip.conf                   # Gzip compression configuration
│   ├── Dockerfile.create-ssl       # Dockerfile for creating SSL
certificates
│   └── Dockerfile.dev              # Dockerfile for the Nginx service in
development
├── site/                           # Contains website files
```

```
│   ├── index.html                  # Main HTML file
│   └── static/                     # Folder for static assets
│       ├── css/
│       │   └── styles.css          # Test CSS file
│       ├── js/
│       │   └── main.js             # Test JavaScript file
│       └── images/                 # (Optional) Folder for image files
├── init-letsencrypt.sh.sample      # Sample SSL creation script
├── init-letsencrypt.sh             # Actual SSL creation script (added by
the user)
├── .gitignore                      # Git ignore file
├── docker-compose-create-ssl.yml   # Docker Compose file for creating SSL
certificates
├── docker-compose-dev.yml          # Docker Compose file for development
setup
```

# Step 04: Running Application on The Server

## Goal of This Step

In this step, we will:

1. Configure Nginx for a production environment with SSL.
2. Create the necessary Docker configurations to serve the app securely.
3. Set up a script to automate the deployment process.
4. Ensure the app is accessible through `https://APP_URL`.

By the end of this step, your app will run in production mode, fully secured with SSL.

---

## Steps to Configure the Production Environment

### 1. Create the Nginx Configuration for Production

1. In the `nginx` folder, create a file named `default-prod-ssl.conf` with the following content:

```
# Redirects HTTP traffic to HTTPS
server {
    # Listens for HTTP traffic on port 80
    listen 80;

    # Replace test4tutorial.work with your actual domain
    server_name test4tutorial.work www.test4tutorial.work;

    # Hides Nginx version info for security
    server_tokens off;

    # Handles Let's Encrypt requests for SSL verification
    location /.well-known/acme-challenge/ {
        # Allows all traffic to this location
```

```
        allow all;

        # Serves the challenge files from this directory
        root /var/www/certbot;
    }

    # Redirects all traffic to HTTPS
    location / {
        return 301 https://$host$request_uri;
    }
}

# Redirects traffic from www.test4tutorial.work to test4tutorial.work over
HTTPS
server {
    # Listens for HTTPS traffic on port 443
    listen 443 ssl;

    # Redirects requests to the non-www version of the domain
    server_name www.test4tutorial.work;

    # SSL certificate and key paths generated by Let's Encrypt
    ssl_certificate
/etc/letsencrypt/live/test4tutorial.work/fullchain.pem;
    ssl_certificate_key
/etc/letsencrypt/live/test4tutorial.work/privkey.pem;

    # Includes SSL configuration provided by Let's Encrypt
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    # Redirects requests to the non-www domain
    return 301 https://test4tutorial.work$request_uri;
}

# Handles HTTPS traffic for test4tutorial.work
server {
    # Listens for HTTPS traffic on port 443
    listen 443 ssl;

    # Replace test4tutorial.work with your actual domain
    server_name test4tutorial.work;

    # SSL certificate and key paths generated by Let's Encrypt, and
Replace test4tutorial.work with your actual domain
    ssl_certificate
/etc/letsencrypt/live/test4tutorial.work/fullchain.pem;
    ssl_certificate_key
/etc/letsencrypt/live/test4tutorial.work/privkey.pem;

    # Includes SSL configuration provided by Let's Encrypt
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
```

```
    # Specifies the root directory for static and HTML files
    root /var/www/app;
    index index.html;

    # Custom 404 error page
    error_page 404 /404.html;

    # Includes Gzip compression configuration
    include /etc/nginx/conf.d/gzip.conf;

    # Serves static files (CSS, JS, images, etc.) from the /static/ folder
    location /static/ {
        alias /var/www/app/static;
    }

    # Configures long-term caching for images
    location ~* \.(gif|jpe?g|png|svg)$ {
        expires max;
        add_header Cache-Control "public, max-age=31536000, immutable";
    }

    # Configures no-cache headers for CSS, JS, and JSON files
    location ~* \.(css|js|json)$ {
        add_header Cache-Control "no-cache, must-revalidate";
    }

    # Serves requested files or falls back to a 404 error
    location / {
        try_files $uri $uri.html $uri/ /404.html;
    }
}
```

---

## 2. Create the Dockerfile for Production

1. In the `nginx` folder, create a file named `Dockerfile.prod-ssl` and add the following content:

```
# Use the lightweight Alpine-based Nginx image as the base image
FROM nginx:1.20.2-alpine

# Copy the production Nginx configuration into the container
COPY ./default-prod-ssl.conf /etc/nginx/conf.d/default.conf

# Copy the Gzip compression configuration
COPY ./gzip.conf /etc/nginx/conf.d/gzip.conf

# Create the static directory in the container
RUN mkdir -p /var/www/app/static && \
    chmod 755 /var/www/app/static
```

---

### 3. Create the Docker Compose File for Production

1. In the root folder of your project, create a file named `docker-compose-prod-ssl.yml` and add the following content:

```
services:
  nginx:
    # Restart the container automatically in case of failure
    restart: always

    # Build the Nginx image using the production Dockerfile
    build:
      context: ./nginx
      dockerfile: Dockerfile.prod-ssl

    # Map ports 80 (HTTP) and 443 (HTTPS) from the container to the
server
    ports:
      - "80:80"
      - "443:443"

    # Mount necessary volumes
    volumes:
      - ./site:/var/www/app # Static files and HTML
      - ./nginx/certbot/www:/var/www/certbot # Let's Encrypt challenge
directory
      - ./nginx/certbot/conf:/etc/letsencrypt # SSL certificates

  certbot:
    # Use the Certbot Docker image for SSL certificate management
    image: certbot/certbot:latest
    container_name: certbot

    # Share volumes for SSL certificates and challenges
    volumes:
      - ./nginx/certbot/www:/var/www/certbot
      - ./nginx/certbot/conf:/etc/letsencrypt

    # Runs Certbot in a loop to renew SSL certificates periodically
    entrypoint: "/bin/sh -c 'trap exit TERM; while :; do sleep 6h &
wait $${!}; certbot renew; done'"
```

---

### 4. Create the Deployment Script

1. In the root folder of your project, create a file named `run_app.sh` and make it executable:

```
chmod +x run_app.sh
```

2. Add the following content to run_app.sh:

```
# Pull the latest changes from the GitHub repository
git pull origin master

# Remove all running and stopped containers
docker container rm -f $(docker container ls -a -q)

# Remove all unused Docker images
docker image rm -f $(docker image ls -a -q)

# Stop and remove the current Docker Compose stack for SSL creation (if
running)
docker-compose -f docker-compose-create-ssl.yml down

# Stop and remove the current Docker Compose stack for production (if
running)
docker-compose -f docker-compose-prod-ssl.yml down

# Build and start the production Docker Compose stack
# Removes unused Docker volumes.
# Removes all unused build cache data.
# Removes all dangling and unused Docker images.
# Removes all stopped containers.
# Removes all unused Docker networks.
docker-compose -f docker-compose-prod-ssl.yml up --build -d && docker
volume prune -f && docker builder prune -a -f && docker image prune -a -f
&& docker container prune -f && docker network prune -f
```

---

## 5. Push the Changes to GitHub

1. Add all the newly created files to the Git repository:

```
git add .
git commit -m "Add production setup with SSL configuration"
git push origin main
```

---

## 6. Deploy the App on the Server

1. **Pull the Latest Changes**:
    - Log into your server and navigate to the `/var/www/app` directory:

```
cd /var/www/app
```

- Pull the latest changes from the GitHub repository:

```
git pull origin main
```

2. **Run the Deployment Script**:

- Execute the deployment script to set up and run the production environment:

```
./run_app.sh
```

---

## 7. Verify Your App

1. **Check App Accessibility**:
   - Visit `https://APP_URL` (replace `APP_URL` with your actual domain name).
   - Verify that your app is accessible and loading over HTTPS.
2. **Confirm SSL Configuration**:
   - Ensure the connection is secure, indicated by a padlock icon in the browser's address bar.
   - Both `https://APP_URL` and `https://www.APP_URL` should be functional.
3. **Verify Static Assets**:
   - Ensure that all static files (e.g., CSS, JavaScript, images) are being served correctly.
   - Check browser developer tools (**Network tab**) to confirm files are served with appropriate caching and compression.

---

## 8. Folder and File Structure

At the end of this step, your project folder structure should look like this:

```
.
├── nginx/                          # Contains Nginx configurations and
Dockerfiles
│   ├── default-prod-ssl.conf       # Nginx configuration for production
with SSL
│   ├── default-create-ssl.conf     # Nginx configuration for creating SSL
certificates
│   ├── default-dev.conf            # Nginx configuration for the
development environment
│   ├── gzip.conf                   # Gzip compression configuration
│   ├── Dockerfile.prod-ssl         # Dockerfile for the Nginx production
setup
│   ├── Dockerfile.create-ssl       # Dockerfile for SSL creation setup
│   └── Dockerfile.dev              # Dockerfile for the Nginx development
```

```
setup
├── site/                           # Contains website files
│   ├── index.html                  # Main HTML file
│   └── static/                     # Folder for static assets
│       ├── css/
│       │   └── styles.css          # Test CSS file
│       ├── js/
│       │   └── main.js             # Test JavaScript file
│       └── images/                 # (Optional) Folder for image files
├── init-letsencrypt.sh.sample      # Sample SSL creation script
├── init-letsencrypt.sh             # Actual SSL creation script (added by
the user)
├── run_app.sh                      # Deployment script for the production
environment
├── .gitignore                      # Git ignore file
├── docker-compose-prod-ssl.yml     # Docker Compose file for production
setup
├── docker-compose-create-ssl.yml   # Docker Compose file for creating SSL
certificates
├── docker-compose-dev.yml          # Docker Compose file for development
setup
```