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

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-1`, `step-2`):
    - 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 1: 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.

---

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.

---

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?