akarsh /
DevOpsLab

Code · Issues · Pull requests · Actions · Projects · Security · Insights · Settings

main · DevOpsLab / README.md

akarsh Rename the README.md file · 646485b · 8 minutes ago

1073 lines (756 loc) · 31 KB

Preview · Code · Blame · Raw

Table of Contents

Experiment 1

Write code for a simple user registration form for an event.

Files

- `app.py` : The main Flask application file that contains the routes and logic for the user registration form.
- `Dockerfile` : The Dockerfile used to build the Docker image for the Flask application.
- `Dockerfile.test` : The Dockerfile used to build the Docker image for running unit tests on the Flask application.
- `requirements.txt` : A file listing the Python dependencies required to run the Flask application.
- `templates/` : A directory containing HTML templates for the Flask application.
    - `register.html` : The HTML template for the user registration form.
    - `success.html` : The HTML template displayed after a successful user registration.

Build

To build the Docker image for the Flask application, run the following command:

```
docker build --no-cache -t simple-flask-app .
```

Run

To run the Docker container for the Flask application, use the following command:

```
docker run -p 80:80 simple-flask-app
```

After running the container, open your web browser and navigate to `http://0.0.0.0:80/register` to access the user registration form.

Testing

To run the unit tests for `app.py` using Docker, follow these steps:

1. Build the Docker image for testing:

```
docker build -f Dockerfile.test -t simple-flask-app-test .
```

2. Run the Docker container to execute the tests:

```
docker run simple-flask-app-test
```

This command will discover and run all the unit tests in the `test_app.py` file.

Experiment 2

Explore Git and GitHub commands

Git

Git is a distributed version control system that helps developers collaborate on projects. It allows you to track changes, revert to previous stages, and branch to create separate lines of development.

GitHub

GitHub is a web-based platform that uses Git for version control. It provides a collaborative environment for developers to host and review code, manage projects, and build software together.

Git commands

- `git init` : Initialize a new Git repository
- `git clone <repository>` : Clone an existing repository
- `git status` : Show the working directory status
- `git add <file>` : Add a file to the staging area
- `git commit -m "message"` : Commit changes with a message
- `git log` : Show the commit history
- `git branch` : List, create, or delete branches
- `git checkout <branch>` : Switch to a different branch
- `git merge <branch>` : Merge a branch into the current branch
- `git pull` : Fetch and merge changes from a remote repository
- `git pull origin <branch>` : Fetch and merge changes from a remote repository branch
- `git push` : Push changes to a remote repository
- `git push origin <branch>` : Push changes to a remote repository branch

About gh

`gh` is GitHub's official command-line tool. It brings GitHub's features to your terminal, allowing you to interact with GitHub repositories, issues, pull requests, and more directly from the command line. This tool helps streamline your workflow by integrating GitHub operations into your existing terminal commands.

gh commands

- `gh auth login` : Authenticate with your GitHub account
- `gh repo create` : Create a new repository on GitHub
- `gh repo clone <repository>` : Clone a GitHub repository
- `gh repo fork <repository>` : Fork a repository on GitHub
- `gh issue list` : List issues in a repository
- `gh issue create` : Create a new issue in a repository
- `gh pr list` : List pull requests in a repository
- `gh pr create` : Create a new pull request
- `gh pr merge <pull-request>` : Merge a pull request

Pull request creation in GitHub

A pull request (PR) is a method of submitting contributions to a project. It allows you to notify project maintainers about changes you'd like them to consider. Here's how to create a pull request in GitHub:

1. **Fork the repository**: Create a copy of the repository under your own GitHub account.
2. **Clone the repository**: Clone the forked repository to your local machine using `git clone <repository-url>` .
3. **Create a new branch**: Create a new branch for your changes using `git checkout -b <branch-name>` .
4. **Make your changes**: Make the necessary changes to the codebase.
5. **Commit your changes**: Commit your changes with a descriptive message using `git commit -m "Your commit message"` .
6. **Push your changes**: Push the changes to your forked repository using `git push origin <branch-name>` .
7. **Create the pull request**: Go to the original repository on GitHub and click on the "New pull request" button. Select the branch you pushed your changes to and create the pull request.

Once the pull request is created, the project maintainers will review your changes and decide whether to merge them into the main codebase.

Experiment 3

Practice Source code management on GitHub. Experiment with the source code in experiment 1.

Steps to Create a Repository on GitHub and Make a Git Commit

1. **Create a Repository on GitHub:**

   - Go to GitHub and log in to your account.
   - Click on the `+` icon in the top right corner and select `New repository` .
   - Enter a repository name (e.g., `Experiment1`), add a description (optional), and choose the visibility (public or private).
   - Click on `Create repository` .

2. **Initialize Git in the `Experiment1` Folder:**

   - Open a terminal and navigate to the `Experiment1` folder:

```
    cd /path/to/Experiment1
    ```

   - Initialize a new Git repository:

     ```
     git init
     ```

3. **Add Remote Repository:**

   - Add the GitHub repository as a remote:

     ```
     git remote add origin https://github.com/your-username/Experiment1.git
     ```

   Note: Replace `your-username` with your actual GitHub username.

4. **Add Files and Make a Commit:**

   - Add all files in the Experiment1 folder to the staging area:

     ```
     git add .
     ```

   - Commit the changes with a message:

     ```
     git commit -m "Initial commit for Experiment1"
     ```

5. **Push Changes to GitHub:**

   - Push the commit to the GitHub repository:

     ```
     git push -u origin master
     ```

You have now created a repository on GitHub and committed the files from the Experiment1 folder.

Experiment 4

Jenkins installation and setup, explore the environment.

About Jenkins

Jenkins is an open-source automation server that helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery (CI/CD).

How to Install Jenkins

Using Docker

1. **Install Docker:** If Docker is not already installed, follow the instructions on the Docker website to install it.

2. **Run Jenkins Container:** Pull the Jenkins image and run it in a container.

   ```
   docker pull jenkins/jenkins:lts
   docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts
   ```

   - `docker pull jenkins/jenkins:lts` : This command pulls the latest stable (LTS) Jenkins image from the Docker repository.
   - `docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts` : This command runs the Jenkins container with the following options:
     - `-p 8080:8080` : Maps port 8080 on the host to port 8080 on the container, allowing access to the Jenkins web interface.
     - `-p 50000:50000` : Maps port 50000 on the host to port 50000 on the container, used for Jenkins agent communication.
     - `-v jenkins_home:/var/jenkins_home` : Mounts the `jenkins_home` volume to persist Jenkins data.

How to Start Jenkins

1. **Access Jenkins:** Open your web browser and go to `http://localhost:8080` . You will see the Jenkins setup wizard.

How to Explore Jenkins

1. **Unlock Jenkins:** Retrieve the initial admin password.

   ```
   docker exec -it <container_id> cat /var/jenkins_home/secrets/initialAdminPassword
   ```

   Enter this password in the setup wizard to unlock Jenkins.

2. **Install Suggested Plugins:** Follow the setup wizard to install the suggested plugins.

3. **Create Admin User:** Create your first admin user as prompted by the setup wizard.

4. **Start Using Jenkins:** Once the setup is complete, you can start creating jobs, configuring pipelines, and exploring Jenkins features.

Experiment 5

Demonstrate continuous integration and development using Jenkins.

Prerequisites

- Docker installed
- Git installed
- A GitHub repository

Setup Jenkins with Docker

1. Build the Docker image:

```
docker build -t jenkins-docker . --platform=linux/amd64
```

2. Run the Docker container:

```
docker run -d -p 8080:8080 -p 50000:50000 --name jenkins-docker jenkins-docker
```

3. Access Jenkins at `http://localhost:8080` and complete the setup process.

Steps

1. **Create a Jenkins Pipeline:**
   - Open Jenkins and create a new pipeline job.
   - In the pipeline configuration, define your pipeline script.
2. **Configure Source Code Management:**
   - Under the Source Code Management section, select Git.
   - Enter the repository URL and credentials if required.
3. **Define Build Steps:**
   - In the pipeline script, define the stages for building, testing, and deploying your application.
4. **Run the Pipeline:**
   - Save the configuration and run the pipeline.
   - Monitor the build process and ensure it completes successfully.
5. **Continuous Integration:**
   - Make changes to your code and push them to the repository.
   - Observe Jenkins automatically triggering the pipeline and running the build.
6. **Continuous Deployment:**
   - Configure the pipeline to deploy the application to a server or cloud service after a successful build.

Example: Using Jenkins to create a pipeline job

Prerequisites

- Jenkins installed and running

Setup

1. Open Jenkins and create a new pipeline job named `Hello World Pipeline` .
2. Add the below pipeline script

Jenkins Pipeline Script

```
pipeline {
    agent any

    stages {
        stage('Hello') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Running the Pipeline

1. Save the pipeline configuration.
2. Run the pipeline and monitor the build process.
3. Make changes to the Experiment1 files and push them to the repository.
4. Observe Jenkins automatically triggering the pipeline and running the build and test stages.

Conclusion

By following these steps, you have demonstrated continuous integration and development using Jenkins.

Experiment 6

Explore Docker commands for content management.

Introduction

In this experiment, we will explore various Docker commands used for content management. Docker is a platform that enables developers to automate the deployment of applications inside lightweight, portable containers.

Objectives

- Understand the basic Docker commands for managing content.
- Learn how to create, manage, and delete Docker images and containers.
- Explore Docker volumes and networks.

Docker Commands for Content Management

1. **docker pull**: Download an image from a Docker registry.

   ```
   docker pull <image_name>
   ```

2. **docker images**: List all Docker images on the local machine.

   ```
   docker images
   ```

3. **docker rmi**: Remove one or more Docker images.

   ```
   docker rmi <image_name>
   ```

4. **docker run**: Create and start a new container from an image.

   ```
   docker run <image_name>
   ```

5. **docker ps**: List all running containers.

   ```
   docker ps
   ```

6. **docker stop**: Stop a running container.

   ```
   docker stop <container_id>
   ```

7. **docker rm**: Remove one or more stopped containers.

   ```
   docker rm <container_id>
   ```

8. **docker volume**: Manage Docker volumes.

   ```
   docker volume ls
   docker volume create <volume_name>
   docker volume rm <volume_name>
   ```

9. **docker network**: Manage Docker networks.

   ```
   docker network ls
   docker network create <network_name>
   docker network rm <network_name>
   ```

Example: Running a Nginx Container

Let's run an example to demonstrate the usage of Docker commands by running an Nginx container.

1. **Pull the Nginx image**:

   ```
   docker pull nginx
   ```

2. **Run the Nginx container**:

   ```
   docker run --name mynginx -d -p 8080:80 nginx
   ```

   This command will start an Nginx container named `mynginx` and map port 8080 on the host to port 80 in the container.

3. **List running containers**:

   ```
   docker ps
   ```

4. **Explore Nginx in the browser**: Open your web browser and navigate to `http://localhost:8080`. You should see the Nginx welcome page.

5. **Stop the Nginx container**:

   ```
   docker stop mynginx
   ```

6. **Remove the Nginx container**:

```
docker rm mynginx
```

7. **Remove the Nginx image**:

   ```
   docker rmi nginx
   ```

Experiment 7

Develop a simple containerized application using Docker

Dockerfile

Create a file named `Dockerfile` with the following content:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Run the hello.py script when the container launches
CMD ["python", "hello.py"]
```

Hello World Python Program

Create a file named `hello.py` with the following content:

```
print("Hello, World!")
```

Build the Docker Image

Run the following command to build the Docker image:

```
docker build -t hello-world-app .
```

Run the Docker Container

Run the following command to start a container from the image:

```
docker run hello-world-app
```

Experiment 8

Integrate Kubernetes and Docker.

Kubernetes and Docker Integration

Kubernetes and Docker are two essential tools in the world of containerization and orchestration. While Docker is a platform for developing, shipping, and running applications inside containers, Kubernetes is an orchestration system for managing containerized applications at scale.

Integration

- **Docker**: Docker allows you to package your application and its dependencies into a container, ensuring that it runs consistently across different environments.
- **Kubernetes**: Kubernetes manages these containers, providing features like automated deployment, scaling, and management of containerized applications.

By integrating Docker with Kubernetes, you can leverage the strengths of both tools to build, deploy, and manage scalable applications efficiently.

Differences

- **Scope**:
  - Docker focuses on the creation and management of individual containers.
  - Kubernetes focuses on the orchestration of multiple containers across a cluster of machines.
- **Components**:
  - Docker includes components like Docker Engine, Docker Compose, and Docker Swarm.
  - Kubernetes includes components like kubectl, kubelet, and kube-scheduler.
- **Scaling**:
  - Docker Swarm provides basic container orchestration and scaling.
  - Kubernetes offers advanced orchestration, scaling, and management capabilities.

Understanding these differences and how they complement each other is crucial for effectively using both tools in your DevOps workflow.

Steps to Integrate Kubernetes and Docker

1. **Install Docker**:
   - Follow the official Docker installation guide for your operating system: Docker Installation
2. **Install Kubernetes (kubectl and minikube)**:
   - Follow the official Kubernetes installation guide: Kubernetes Installation
3. **Start Minikube**:

   ```
   minikube start
   ```

4. **Build a Docker Image**:
   - Create a Dockerfile for your application.
   - Build the Docker image:

   ```
   docker build -t your-image-name .
   ```

5. **Push Docker Image to a Registry**:
   - Tag your Docker image:

   ```
   docker tag your-image-name your-dockerhub-username/your-image-name
   ```

   - Push the image to Docker Hub:

   ```
   docker push your-dockerhub-username/your-image-name
   ```

6. **Create a Kubernetes Deployment**:
   - Create a deployment YAML file (e.g., `deployment.yaml`):

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: your-deployment-name
   spec:
     replicas: 2
     selector:
       matchLabels:
         app: your-app-name
     template:
       metadata:
         labels:
           app: your-app-name
       spec:
         containers:
           - name: your-container-name
             image: your-dockerhub-username/your-image-name
             ports:
               - containerPort: 80
   ```

   - Apply the deployment:

   ```
   kubectl apply -f deployment.yaml
   ```

7. **Expose the Deployment**:
   - Create a service YAML file (e.g., `service.yaml`):

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: your-service-name
   spec:
     type: LoadBalancer
     selector:
       app: your-app-name
     ports:
       - protocol: TCP
         port: 80
         targetPort: 80
   ```

   - Apply the service:

   ```
   kubectl apply -f service.yaml
   ```

8. **Access Your Application**:
   - Get the URL of your application:

   ```
   minikube service your-service-name --url
   ```

   - Open the URL in your browser to access your application.

Experiment 9

Automate the process of running containerized application developed

in experiment 7 using kubernetes

Introduction

In this experiment, we will automate the deployment of a containerized application using Kubernetes. This involves creating Kubernetes manifests for the application and deploying it to a Kubernetes cluster.

Prerequisites

- Docker installed
- Minikube installed and running
- kubectl installed and configured
- Ensure you have the necessary permissions to create resources in the Kubernetes cluster

Steps

1. Start Minikube

Start Minikube to create a local Kubernetes cluster:

```
minikube start
```

2. Set Minikube Context

Set Minikube as the Kubernetes context:

```
kubectl config use-context minikube
```

3. Create and Expose Deployment

Create a sample deployment and expose it on port 8080:

```
kubectl create deployment hello-minikube --image=kicbase/echo-server:1.0
kubectl expose deployment hello-minikube --type=NodePort --port=8080
```

It may take a moment, but your deployment will soon show up when you run:

```
kubectl get services hello-minikube
```

The easiest way to access this service is to let Minikube launch a web browser for you:

```
minikube service hello-minikube
```

4. Check Minikube Dashboard

You can also check the Minikube dashboard to monitor your cluster:

```
minikube dashboard
```

Conclusion

By following these steps, you have automated the deployment of a containerized application using Kubernetes with Minikube.

Experiment 10

Install and explore selenium for automated testing

Using Python, Selenium, and Docker for Testing

1. **Install Docker**: Follow the instructions on the [Docker website]() to install Docker on your machine.

2. **Create Docker Compose File**: Create a `docker-compose.yml` file to set up services for Python and Selenium.

```
version: "3"
services:
  selenium:
    image: selenium/standalone-firefox:latest
    ports:
      - "4444:4444"
  test:
    image: python:3.8-slim
    volumes:
      - .:/app
    working_dir: /app
    depends_on:
      - selenium
    entrypoint: ["sh", "-c", "apt-get update && apt-get install -y netcat-openbsd && while ! nc -z selenium 4444;
```

3. **Write a Selenium Test Script**: Create a Python script `test_script.py` with Selenium tests.

```
from selenium import webdriver
from selenium.webdriver.firefox.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.firefox.options import Options

# Set up the Firefox WebDriver
options = Options()
options.headless = True
driver = webdriver.Remote(
    command_executor='http://selenium:4444/wd/hub',
    options=options
)

# Test: Open Selenium website and check the page title
driver.get("https://www.selenium.dev")
assert "Selenium" in driver.title

# Print success message
print("Test passed: Selenium website title is correct.")

driver.quit()
```

4. **Run the Docker Compose Services**:

```
docker compose up --build
```

Explanation of the Test and Output

- The test script uses Selenium to open the Selenium website and check if the page title contains the word "Selenium".
- If the test passes, a success message "Test passed: Selenium website title is correct." is printed to the console.
- When you run `docker compose up --build`, Docker will build and start the services defined in the `docker-compose.yml` file.
- The `test` service will wait for the `selenium` service to be ready, install the necessary dependencies, and then execute the test script.
- You will see the output of the test in the Docker console, including the success message if the test passes.

Experiment 11

Write a simple program in JavaScript and perform testing using Selenium

Using Docker Compose for Selenium Firefox to Test HTML, CSS, and JS

1. **Install Docker**: Follow the instructions on the Docker website to install Docker on your machine.

2. **Create Docker Compose File**: Create a `docker-compose.yml` file to set up services for Python and Selenium.

```
services:
  selenium:
    image: selenium/standalone-firefox:latest
    ports:
      - "4444:4444"
    volumes:
      - .:/app
  test:
    image: python:3.8-slim
    volumes:
      - .:/app
    working_dir: /app
    depends_on:
      - selenium
    entrypoint:
      [
        "sh",
        "-c",
        "apt-get update && apt-get install -y netcat-openbsd && while ! nc -z selenium 4444; do sleep 1; done &&
      ]
```

3. **Create HTML, CSS, and JS Files**: Create a simple HTML file `index.html` that shows "Hello World" and changes the color via CSS.

```
<!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="styles.css" />
  </head>
  <body>
    <h1 id="hello">Hello World</h1>
    <script src="script.js"></script>
  </body>
</html>
```

4. **Create a CSS File**: Create a CSS file `styles.css` to style the webpage.

```
body {
  font-family: Arial, sans-serif;
  text-align: center;
  margin-top: 50px;
}

#hello {
  color: blue;
}
```

5. **Create a JavaScript File**: Create a JavaScript file `script.js` to add color to the webpage.

```
document.getElementById("hello").style.color = "blue";
```

6. **Write a Selenium Test Script**: Create a Python script `test_script_docker.py` to test the HTML file.

```
from selenium import webdriver
from selenium.webdriver.firefox.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.firefox.options import Options
import os
import time  # Added import for time

# Check if index.html file exists
file_path = "/app/index.html"
if not os.path.exists(file_path):
    raise FileNotFoundError(f"{file_path} does not exist")
else:
    print(f"Found {file_path}")

# Set up the Firefox WebDriver
options = Options()
options.headless = True
driver = webdriver.Remote(
    command_executor='http://selenium:4444/wd/hub',
    options=options
)

# Wait for the Selenium server to be ready
time.sleep(1)

try:
    # Test: Open the local HTML file and check the content and color
    abs_file_path = os.path.abspath(file_path)
    print(f"Absolute file path: {abs_file_path}")
    driver.get(f"file://{abs_file_path}")
    element = driver.find_element(By.ID, "hello")
    assert element.text == "Hello World"
    assert element.value_of_css_property("color") == "rgb(0, 0, 255)"  # blue color

    # Print success message
    print("Test passed: 'Hello World' is displayed correctly with blue color.")
finally:
    driver.quit()  # Ensure the WebDriver quits even if an assertion fails
```

7. **Run the Docker Compose Services**:

```
docker compose up --build
```

Explanation of the Test and Output

- The test script uses Selenium to open the local HTML file and check if the content of the element with ID "hello" is "Hello World" and if its color is blue.
- If the test passes, a success message "Test passed: 'Hello World' is displayed correctly with blue color." is printed to the console.
- When you run `docker compose up --build`, Docker will build and start the services defined in the `docker-compose.yml` file.
- The `test` service will wait for the `selenium` service to be ready, install the necessary dependencies, and then execute the test script.
- You will see the output of the test in the Docker console, including the success message if the test passes.

Experiment 12

Develop test cases for the above containerized application using selenium

Using Docker Compose for Selenium Firefox to Test HTML, CSS, and JS

1. **Install Docker**: Follow the instructions on the Docker website to install Docker on your machine.

2. **Create Docker Compose File**: Create a `docker-compose.yml` file to set up services for Python and Selenium.

```
services:
  selenium:
    image: selenium/standalone-firefox:latest
    ports:
      - "4444:4444"
    volumes:
      - .:/app
  test:
    image: python:3.8-slim
    volumes:
      - .:/app
    working_dir: /app
    depends_on:
      - selenium
    entrypoint:
      [
        "sh",
        "-c",
        "apt-get update && apt-get install -y netcat-openbsd && while ! nc -z selenium 4444; do sleep 1; done &&
      ]
```

3. **Create HTML, CSS, and JS Files**: Create a simple HTML file `index.html` that shows "Hello World" and changes the color via CSS.

```
<!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="styles.css" />
  </head>
  <body>
    <h1 id="hello">Hello World</h1>
    <button id="counterButton">Click me</button>
    <p id="counterDisplay">0</p>
    <script src="script.js"></script>
  </body>
</html>
```

4. **Create a CSS File**: Create a CSS file `styles.css` to style the webpage.

```
body {
  font-family: Arial, sans-serif;
  text-align: center;
  margin-top: 50px;
}

#hello {
  color: blue;
}
```

5. **Create a JavaScript File**: Create a JavaScript file `script.js` to add color to the webpage and handle the counter functionality.

```
document.addEventListener("DOMContentLoaded", function () {
  document.getElementById("hello").style.color = "blue";

  let counter = 0;
  const counterButton = document.getElementById("counterButton");
  const counterDisplay = document.getElementById("counterDisplay");

  counterButton.addEventListener("click", function () {
    counter++;
    counterDisplay.textContent = counter;
  });
});
```

6. **Write a Selenium Test Script**: Create a Python script `test_script_docker.py` to test the HTML file.

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.firefox.options import Options
import os
import time
import unittest

# Check if index.html file exists
file_path = "/app/index.html"
if not os.path.exists(file_path):
    raise FileNotFoundError(f"{file_path} does not exist")
else:
    print(f"Found {file_path}")

# Set up the Firefox WebDriver
options = Options()
options.headless = True

class TestWebPage(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.driver = webdriver.Remote(
            command_executor='http://selenium:4444/wd/hub',
            options=options
        )
        cls.driver.get(f"file://{os.path.abspath(file_path)}")
        time.sleep(1)  # Wait for the page to load

    @classmethod
    def tearDownClass(cls):
        cls.driver.quit()

    def test_find_html_element(self):
        try:
            hello_element = self.driver.find_element(By.ID, "hello")
            self.assertIsNotNone(hello_element)
            self.assertEqual(hello_element.text, "Hello World")
            self.assertEqual(hello_element.value_of_css_property("color"), "rgb(0, 0, 255)")  # blue color
            print("Test passed: 'Hello World' element found with correct text and color.")
        except AssertionError as e:
            print(f"Test failed: {e}")

    def test_counter_increment(self):
        try:
            counter_button = self.driver.find_element(By.ID, "counterButton")
            counter_display = self.driver.find_element(By.ID, "counterDisplay")

            initial_value = int(counter_display.text)
            counter_button.click()
            updated_value = int(counter_display.text)

            self.assertEqual(updated_value, initial_value + 1)
            print("Test passed: Counter incremented correctly.")
        except AssertionError as e:
            print(f"Test failed: {e}")

if __name__ == "__main__":
    unittest.main()
```

7. **Run the Docker Compose Services**:

```
docker compose up --build
```

Explanation of the Test and Output

- The test script uses Selenium to open the local HTML file and check if the content of the element with ID "hello" is "Hello World" and if its color is blue.
- It also tests if the counter increments correctly when the button is clicked.
- If the tests pass, success messages are printed to the console.
- When you run `docker compose up --build`, Docker will build and start the services defined in the `docker-compose.yml` file.
- The `test` service will wait for the `selenium` service to be ready, install the necessary dependencies, and then execute the test script.
- You will see the output of the tests in the Docker console, including the success messages if the tests pass.

New Features

- Added a button that increments a counter each time it is clicked.
- The counter value is displayed below the button.

Test Cases

1. Test case to find the HTML element.
2. Test case to update the counter and test it.