I N D U R

INSTITUTE OF ENGINEERING AND TECHNOLOGY
SIDDIPET, MEDAK DIST. – 502277

LABORATORY MANUAL

DEVOPS

III B.Tech. CSE
(As per R22 Academic Regulation)

DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING

PREFACE

This manual is designed to accompany the practical sessions of the DevOps course and aims to provide you with hands-on experience in implementing DevOps practices.

In today's rapidly evolving IT industry, the need for continuous integration, continuous delivery, and automation has become imperative for organizations to stay competitive. DevOps, a set of practices that combine software development (Dev) and IT operations (Ops), facilitates seamless collaboration between development and operations teams to deliver high-quality software products at a faster pace.

This laboratory course is structured to equip you with the fundamental principles, tools, and techniques of DevOps. Through a series of practical exercises, you will learn to set up development environments, automate software deployment, monitor system performance, and implement continuous integration and continuous delivery pipelines.

The curriculum covers a wide range of topics, including version control systems (such as Git), containerization (using Docker), configuration management (with tools like Ansible), continuous integration (using Jenkins), infrastructure as code (with Terraform), and orchestration (using Kubernetes). Each topic is accompanied by hands-on exercises to reinforce your understanding and proficiency.

This manual serves as your guide throughout the lab sessions, providing detailed instructions, explanations, and references to supplementary resources. It is intended to support your learning journey and help you develop the practical skills and knowledge required to excel in the field of DevOps.

We encourage you to approach the laboratory sessions with enthusiasm, curiosity, and a willingness to experiment. Don't hesitate to ask questions, seek clarification, and engage actively with your peers and instructors. Remember, the skills you acquire in this course will not only enhance your academic knowledge but also prepare you for real-world challenges in the dynamic landscape of software development and operations.

Course Objectives: The main objectives of this course are to:

1. Describe the agile relationship between development and IT operations.
2. Understand the skill sets and high-functioning teams involved in DevOps and related methods to reach a continuous delivery capability.
3. Implement automated system update and DevOps lifecycle.

Course Outcomes: On successful completion of this course, students will be able to:

1. Identify components of Devops environment.
2. Describe Software development models and architectures of DevOps.
3. Apply different project management, integration, testing and code deployment tool.
4. Investigate different DevOps Software development models.
5. Assess various Devops practices.
6. Collaborate and adopt Devops in real-time projects.

LAB CODE

1. Students should report to the concerned labs as per the time table schedule.
2. Students who turn up late to the labs will in no case be permitted to perform the experiment scheduled for the day.
3. After completion of the experiment, certification of the concerned staff in-charge in the observation book is necessary.
4. Staff member in-charge shall award marks based on continuous evaluation for each Experiment out of maximum 10 marks and should be entered in the notebook
5. Students should bring a note book of about 100 pages and should enter the readings (or) observations into the note book while performing the experiment.
6. The record of observations along with the detailed experimental procedure of the experiment performed in the immediate last session should be submitted and certified by the staff member in-charge.
7. Not more than three students in a group are permitted to perform the experiment on a setup.

8. The group-wise division made in the beginning should be adhered to, and no mix up of student among different groups will be permitted later.

9. The components required pertaining to the experiment should be collected from stores in-charge after duly filling in the requisition form.

10. When the experiment is completed, students should disconnect the setup made by them, and should return all the components/instruments taken for the purpose.

11. Any damage of the equipment or burn-out of components will be viewed seriously either by putting penalty or by dismissing the total group of students from the lab for the semester/year.

12. Students should be present in the labs for the total scheduled duration.

13. Students are required to prepare thoroughly to perform the experiment before coming to Laboratory.

14. Procedure sheets/data sheets provided to the students' groups should be maintained neatly and to be returned after the experiment.

INDEX

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

Here's a simple example of HTML code for a user registration form:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>User Registration Form</title>
    <style>
        form {
            margin: 20px auto;
            width: 300px;
            border: 1px solid #ccc;
            padding: 20px;
            border-radius: 5px;
        }
        input[type="text"], input[type="email"], input[type="submit"] {
            width: 100%;
            margin-bottom: 10px;
            padding: 8px;
            box-sizing: border-box;
            border-radius: 3px;
            border: 1px solid #ccc;
        }
        input[type="submit"] {
            background-color: #4CAF50;
            color: white;
            cursor: pointer;
        }
        input[type="submit"]:hover {
            background-color: #45a049;
        }
    </style>
</head>
<body>
```

```
<form action="submit.php" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>

    <input type="submit" value="Register">
</form>

</body>
</html>
```

This form includes fields for the user's name and email, and a submit button. When the form is submitted, the data is sent to a server-side script specified in the `action` attribute of the `<form>` tag (in this case, `submit.php`) for processing. You would need to create the `submit.php` file to handle the form submission and perform any necessary actions, such as saving the user's information to a database.

2. Explore Git and GitHub commands.

Git is a version control system that allows you to track changes in your codebase over time. GitHub is a hosting service for Git repositories, where you can store your code, collaborate with others, and manage your project's development workflow. Here are some common Git and GitHub commands:

Basic Git Commands:

1. **git init**: Initialize a new Git repository in the current directory.
2. **git clone [url]**: Clone an existing Git repository from a remote URL.
3. **git add [file]**: Add a file or directory to the staging area.
4. **git commit -m "message"**: Commit changes to the repository with a descriptive message.
5. **git status**: Check the status of files in the working directory and staging area.
6. **git log**: View the commit history.
7. **git branch**: List all branches in the repository.
8. **git checkout [branch]**: Switch to a different branch.
9. **git merge [branch]**: Merge changes from a different branch into the current branch.
10. **git pull**: Fetch changes from the remote repository and merge them into the current branch.
11. **git push**: Push local commits to the remote repository.

GitHub-specific Commands:

1. **git remote**: List remote repositories.
2. **git remote add origin [url]**: Add a remote repository named "origin" with the specified URL.
3. **git push -u origin [branch]**: Push the current branch to the remote repository and set it as the upstream branch.
4. **git pull origin [branch]**: Fetch changes from the remote repository and merge them into the current branch.
5. **git clone [repository_url]**: Clone a repository from GitHub to your local machine.
6. **git fork**: Create a copy of a repository on your GitHub account.
7. **git pull-request**: Create a pull request to merge changes from one branch into another.

Workflow Commands:

1. **git branch [branch_name]**: Create a new branch.
2. **git checkout -b [branch_name]**: Create a new branch and switch to it.
3. **git merge [branch_name]**: Merge changes from one branch into another.
4. **git rebase [branch_name]**: Reapply commits from one branch on top of another.
5. **git stash**: Temporarily save changes that are not ready to be committed.

These are just a few of the most common commands used in Git and GitHub. There are many more commands and options available depending on your specific use case and workflow. It's a good idea to

explore the documentation and practice using these commands in a safe environment to become more familiar with them.

3. Practice Source code management on GitHub. Experiment with code for a simple user registration form for an event.

Here's a simple user registration form code that we can create and manage using GitHub:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>User Registration Form</title>
</head>
<body>

<form action="submit.php" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required><br><br>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required><br><br>

    <input type="submit" value="Register">
</form>

</body>
</html>
```

Now, let's go through the steps to manage this code on GitHub:

1. **Create a GitHub Account**: If you don't have one already, sign up for a GitHub account at github.com.

2. **Create a New Repository**:
   - Click on the "+" icon in the top-right corner of the GitHub interface and select "New repository".
   - Choose a name for your repository (e.g., "user-registration-form").
   - Optionally, add a description and choose whether the repository will be public or private.
   - Click "Create repository".
3. **Clone the Repository**:
   - Copy the URL of your newly created repository.
   - Open your terminal or Git Bash.
   - Navigate to the directory where you want to store your project.
   - Run the following command, replacing `[repository_url]` with the URL you copied:

git clone [repository_url]

4. **Add Your Code**:
   - Copy the HTML code for the user registration form into a new file named `index.html` within your local repository.
5. **Commit Your Changes**:
   - In your terminal, navigate to your project directory.
   - Use the following commands to add and commit your changes:

git add index.html

git commit -m "Add user registration form"

6. **Push Your Changes to GitHub**:
- Use the following command to push your changes to GitHub:

git push origin master

   - You may need to replace `master` with `main` depending on the default branch name.
7. **Check Your Repository on GitHub**:
   - Refresh your repository page on GitHub, and you should see your `index.html` file listed.

Now, you've successfully managed your simple user registration form code on GitHub! You can continue to make changes, commit them, and push them to GitHub as needed.

4. Jenkins installation and setup, explore the environment.

Installing and setting up Jenkins involves a few steps, but it's relatively straightforward. Here's a basic guide to installing Jenkins and exploring its environment:

Step 1: Install Jenkins

1. **Download Jenkins**: Go to the official Jenkins website (https://www.jenkins.io/download/) and download the appropriate package for your operating system (e.g., WAR file, installer for Windows, package manager for Linux).
2. **Install Jenkins**:
   - For Windows: Double-click the installer and follow the installation instructions.
   - For Linux: Use your package manager (e.g., `apt`, `yum`, `dnf`) to install Jenkins.
3. **Start Jenkins**:
   - For Windows: Jenkins may start automatically after installation. If not, you can start it from the Start menu.
   - For Linux: Use the following command to start Jenkins:

```
sudo systemctl start Jenkins
```

Step 2: Access Jenkins Web Interface

1. Open a web browser and navigate to `http://localhost:8080` (replace `localhost` with the hostname or IP address of your server if Jenkins is installed remotely).
2. Follow the instructions to complete the setup wizard. You'll need to provide an initial admin password, which can be found in the Jenkins logs or in the file specified during installation.
3. Install suggested plugins or choose which plugins to install manually.
4. Create the first admin user and click "Save and Finish".
5. Start using Jenkins!

Step 3: Explore Jenkins Environment

1. **Dashboard**: After logging in, you'll see the Jenkins dashboard. This is the main hub where you can view all your Jenkins jobs, manage nodes, and access system configuration.
2. **Manage Jenkins**: Click on "Manage Jenkins" on the left sidebar to access various configuration options, such as configuring global tools, managing plugins, and configuring security.
3. **Create a New Job**: Click on "New Item" to create a new Jenkins job. Jobs represent the work you want Jenkins to perform, such as building a project, running tests, or deploying applications.
4. **Configure Jobs**: Once you've created a job, you can configure it by specifying the source code repository, build triggers, build steps, post-build actions, and more.
5. **Build History**: Jenkins keeps a history of all builds performed for each job. You can view details of each build, including the console output, status, and artifacts produced.

6. **Plugins**: Jenkins has a vast ecosystem of plugins that extend its functionality. You can browse and install plugins from the "Manage Jenkins" > "Manage Plugins" section.
7. **User Management**: Jenkins allows you to manage users and their permissions. You can define roles, assign permissions, and control access to various Jenkins features.
8. **Pipeline**: Jenkins Pipeline is a powerful way to define complex automation workflows using a domain-specific language (DSL). You can create and manage pipelines to automate your entire software delivery process.

By exploring these features, you'll gain a better understanding of how Jenkins works and how you can leverage it to automate your software development and delivery processes.

5.Demonstrate continuous integration and development using Jenkins.

Let's walk through a simple example of setting up continuous integration and development (CI/CD) using Jenkins. In this example, we'll create a basic Jenkins job that builds a sample web application from a Git repository and deploys it to a test server.

Prerequisites:

1. Jenkins installed and running.
2. Git installed on the Jenkins server.

Step 1: Create a Jenkins Job

1. Log in to the Jenkins dashboard.
2. Click on "New Item" to create a new Jenkins job.
3. Enter a name for your job (e.g., "SampleAppCI/CD") and select "Freestyle project", then click "OK".

Step 2: Configure the Jenkins Job

1. In the job configuration page, under the "General" section:
   - Check "GitHub project" and enter the URL of your GitHub repository.
   - Specify the branch(es) you want Jenkins to build (e.g., `*/main`).
2. Under the "Source Code Management" section:
   - Choose "Git".
   - Enter the URL of your Git repository.
   - Optionally, specify the credentials to access the repository.
3. Under the "Build Triggers" section:
   - Check "Poll SCM" and set a schedule (e.g., `* * * * *` to poll every minute).
4. Under the "Build" section:
   - Click on "Add build step" and choose "Execute shell" (or "Execute Windows batch command" for Windows).
   - Enter the build commands necessary to build your application. For example, if you're building a Node.js application, you might run `npm install` and `npm run build`.
5. Under the "Post-build Actions" section:
   - Click on "Add post-build action" and choose "Deploy war/ear to a container" (you may need to install the "Deploy to container Plugin" if it's not already installed).
   - Configure the settings to deploy your built artifact (e.g., WAR file) to your test server.

Step 3: Save and Run the Jenkins Job

1. Click on "Save" to save your Jenkins job configuration.

2. Click on "Build Now" to trigger a build manually or wait for the next scheduled build.

Step 4: Monitor the CI/CD Process

1. Jenkins will automatically trigger a build according to the configured schedule or when changes are detected in the Git repository.
2. You can monitor the build progress and view the console output to see the build and deployment steps.
3. If the build is successful, Jenkins will deploy the artifact to your test server.

Step 5: Continuous Integration and Development

1. Whenever changes are pushed to the Git repository, Jenkins will automatically detect them and trigger a new build.
2. The CI/CD pipeline will build the updated code, run tests, and deploy the artifact to the test server.
3. This allows you to continuously integrate and deploy changes to your application, ensuring that it remains stable and up-to-date.

By following these steps, you've set up a basic CI/CD pipeline using Jenkins, allowing you to automate the build, test, and deployment process of your applications. You can further enhance this pipeline by adding more build steps, integrating with additional tools, and configuring more advanced deployment strategies.

6.Explore Docker commands for content management.

Docker provides a set of commands for managing containers, images, volumes, networks, and other resources. Here are some essential Docker commands for content management:

Managing Containers:

1. **docker run**: Create and start a new container from an image.
   docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
2. **docker start**: Start one or more stopped containers.
   docker start [OPTIONS] CONTAINER [CONTAINER...]
3. **docker stop**: Stop one or more running containers.
   docker stop [OPTIONS] CONTAINER [CONTAINER...]
4. **docker restart**: Restart one or more containers.
   docker restart [OPTIONS] CONTAINER [CONTAINER...]
5. **docker rm**: Remove one or more containers.
   docker rm [OPTIONS] CONTAINER [CONTAINER...]
6. **docker ps**: List running containers.
   docker ps [OPTIONS]
7. **docker logs**: Fetch the logs of a container.
   docker logs [OPTIONS] CONTAINER

Managing Images:

1. **docker pull**: Pull an image or a repository from a registry.
   docker pull [OPTIONS] NAME[:TAG|@DIGEST]
2. **docker build**: Build an image from a Dockerfile.
   docker build [OPTIONS] PATH | URL | -
3. **docker push**: Push an image or a repository to a registry.
   docker push [OPTIONS] NAME[:TAG]

4. **docker images**: List images.
   docker images [OPTIONS] [REPOSITORY[:TAG]]

5. **docker rmi**: Remove one or more images.
   docker rmi [OPTIONS] IMAGE [IMAGE...]

Managing Volumes:

1. **docker volume create**: Create a volume.
   docker volume create [OPTIONS] [VOLUME]

2. **docker volume ls**: List volumes.
   docker volume ls [OPTIONS]

3. **docker volume rm**: Remove one or more volumes.
   docker volume rm [OPTIONS] VOLUME [VOLUME...]

Managing Networks:

1. **docker network create**: Create a network.
   docker network create [OPTIONS] NETWORK

2. **docker network ls**: List networks.
   docker network ls [OPTIONS]

3. **docker network rm**: Remove one or more networks.
   docker network rm [OPTIONS] NETWORK [NETWORK...]

These are just some of the most commonly used Docker commands for managing containers, images, volumes, and networks. Docker provides many more commands and options for managing Docker resources. You can explore the Docker documentation for more information on additional commands and their usage.

7.Develop a simple containerized application using Docker.

let's create a simple containerized application using Docker. We'll create a basic Node.js web server that serves a "Hello, World!" message.

Step 1: Create the Node.js Application

First, create a directory for your project and navigate into it:

```
mkdir docker-nodejs-app
cd docker-nodejs-app
```

Create a file named `app.js` and add the following code to it:

```
const http = require('http');

const hostname = '0.0.0.0';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

This code creates a simple HTTP server that listens on port 3000 and responds with "Hello, World!" to any incoming requests.

Step 2: Create a Dockerfile

Next, create a file named `Dockerfile` in the same directory and add the following content to it:

```
FROM node:14
```

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "app.js"]

This Dockerfile sets up a Node.js environment, copies the application files into the container, installs dependencies, exposes port 3000, and starts the Node.js server.

Step 3: Build the Docker Image

Open a terminal in the project directory and run the following command to build the Docker image:

docker build -t docker-nodejs-app .

This command builds a Docker image with the tag `docker-nodejs-app` using the Dockerfile and application files in the current directory.

Step 4: Run the Docker Container

After the image is built, you can run a container based on it using the following command:

docker run -p 3000:3000 docker-nodejs-app

This command starts a container based on the `docker-nodejs-app` image and maps port 3000 of the container to port 3000 on the host system.

Step 5: Test the Application

Open a web browser and navigate to `http://localhost:3000` or use curl:

curl http://localhost:3000

You should see the message "Hello, World!" displayed, indicating that the Node.js application is running inside the Docker container.

That's it! You've successfully created a simple containerized application using Docker. You can extend and customize this example to build more complex applications with Docker.

8. Integrate Kubernetes and Docker.

Integrating Kubernetes with Docker involves using Kubernetes to orchestrate and manage Docker containers at scale. Kubernetes provides features such as container scheduling, scaling, load balancing, and self-healing, while Docker is used to build, package, and run containers.

Here's a high-level overview of how to integrate Kubernetes with Docker:

Step 1: Set up Kubernetes Cluster

1. Install Kubernetes: Set up a Kubernetes cluster using a tool like Minikube for local development or a managed Kubernetes service like Amazon EKS, Google Kubernetes Engine (GKE), or Azure Kubernetes Service (AKS) for production deployments.

Step 2: Containerize Your Application

1. Dockerize your application: Use Docker to containerize your application by creating a Dockerfile and building a Docker image that contains your application and its dependencies.

Step 3: Deploy Your Application to Kubernetes

1. Create Kubernetes Deployment: Write a Kubernetes Deployment manifest (YAML file) that defines how your application should be deployed. This includes specifying the Docker image, ports, replicas, and any other configuration.
2. Apply Deployment: Use the `kubectl apply` command to apply the Deployment manifest and deploy your application to the Kubernetes cluster.

Step 4: Scale and Manage Your Application

1. Scaling: Use Kubernetes to scale your application horizontally by adjusting the number of replicas in the Deployment manifest or using the `kubectl scale` command.
2. Service Discovery and Load Balancing: Kubernetes provides built-in service discovery and load balancing for your application. Use Kubernetes Services to expose your application internally or externally and automatically load balance traffic to your application pods.

Step 5: Monitoring and Logging

1. Monitor Your Application: Set up monitoring and logging for your application using Kubernetes-native tools like Prometheus for monitoring and Grafana for visualization, or integrate with third-party monitoring solutions.

Step 6: Continuous Deployment

1. Set up Continuous Deployment (CD): Integrate Kubernetes with your CI/CD pipeline to automate the deployment of new versions of your application. Tools like Jenkins, GitLab CI/CD, or Argo CD can be used for this purpose.

Step 7: Update and Rollback

1. Update Your Application: Use Kubernetes rolling updates to deploy new versions of your application with zero downtime. Update the Docker image version in the Deployment manifest and apply the changes using `kubectl apply`.
2. Rollback: If a deployment fails or introduces issues, Kubernetes allows you to roll back to a previous version of your application using the `kubectl rollout undo` command.

By integrating Kubernetes with Docker, you can leverage the power of containerization and container orchestration to deploy and manage your applications efficiently and at scale.

9. Automate the process of running containerized application developed in exercise 7 using Kubernetes.

To automate the process of running a containerized application using Kubernetes, we'll create Kubernetes manifests (YAML files) to define the necessary resources like Deployments, Services, and optionally Ingress for exposing the application externally. Let's go through the steps:

Step 1: Define Deployment Manifest

Create a file named `deployment.yaml` and add the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: docker-nodejs-app
spec:
  replicas: 3  # You can adjust the number of replicas as needed
  selector:
    matchLabels:
      app: docker-nodejs-app
  template:
    metadata:
      labels:
        app: docker-nodejs-app
    spec:
      containers:
      - name: docker-nodejs-app
        image: your-dockerhub-username/docker-nodejs-app:latest  # Replace
with your Docker image
        ports:
        - containerPort: 3000
```

Step 2: Define Service Manifest

Create a file named `service.yaml` and add the following content:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: docker-nodejs-app
spec:
  selector:
    app: docker-nodejs-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: ClusterIP  # Change to NodePort or LoadBalancer for external access if needed
```

Step 3: Apply Kubernetes Manifests

Apply the deployment and service manifests to your Kubernetes cluster using the following commands:

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

Step 4: Access the Application

If you used `ClusterIP` type for the service, you can access the application internally within the cluster. If you want to expose the application externally, consider using a `NodePort` or `LoadBalancer` type service, or set up an Ingress resource.

Step 5: Scale Your Application (Optional)

You can scale your application by adjusting the number of replicas in the deployment manifest or using the following command:

```
kubectl scale deployment docker-nodejs-app --replicas=5
```

Step 6: Monitor and Manage Your Application

Use Kubernetes dashboard or command-line tools like `kubectl` to monitor and manage your application. You can check the status of your deployments, pods, services, and other resources using `kubectl get` commands.

Step 7: Continuous Deployment (Optional)

Integrate Kubernetes deployment manifests with your CI/CD pipeline to automate the deployment process whenever you push changes to your application codebase.

By following these steps, you've automated the process of running your containerized application using Kubernetes. This setup allows you to efficiently manage, scale, and deploy your application in a Kubernetes cluster.

10. Install and Explore Selenium for automated testing.

Selenium is a powerful tool for automated testing of web applications. It allows you to automate interactions with web browsers to test various aspects of your web application's functionality, including UI testing, regression testing, and cross-browser testing.

Here's how you can install and explore Selenium:

Step 1: Install Selenium WebDriver

You can install Selenium WebDriver for your preferred programming language. Selenium supports multiple programming languages like Java, Python, JavaScript (Node.js), C#, and Ruby.

Python Example:

You can install Selenium WebDriver for Python using pip:

```
pip install selenium
```

Step 2: Download Browser Drivers

Selenium WebDriver requires browser drivers to communicate with web browsers. Download the appropriate browser driver for the browser you want to automate testing with.

For example, for Chrome, you can download the ChromeDriver from the official website: ChromeDriver - WebDriver for Chrome

Step 3: Write Your Selenium Test Scripts

Now, you can write your test scripts using Selenium WebDriver APIs. Here's a simple example using Python:

```
from selenium import webdriver

# Path to the browser driver executable
chrome_driver_path = '/path/to/chromedriver'

# Create a new instance of Chrome WebDriver
driver = webdriver.Chrome(executable_path=chrome_driver_path)

# Open the website to test
driver.get('https://example.com')
```

```
# Perform actions such as clicking buttons, filling forms, etc.
# Example:
# element = driver.find_element_by_xpath('//button[@id="myButton"]')
# element.click()

# Close the browser
driver.quit()
```

Step 4: Run Your Selenium Test Scripts

Save your test script to a file (e.g., `test_script.py`) and run it using your preferred Python interpreter:

```
python test_script.py
```

Step 5: Explore Selenium Features

You can explore various features of Selenium WebDriver to automate different aspects of your web application, including:

- Locating elements using various locators like ID, name, class name, XPath, CSS selectors, etc.
- Interacting with web elements (clicking buttons, filling forms, submitting forms, etc.).
- Handling alerts, pop-ups, and iframes.
- Performing actions like mouse hover, drag-and-drop, etc.
- Capturing screenshots and handling cookies.
- Writing test suites and integrating with testing frameworks like pytest, unittest, etc.

By exploring these features, you can create comprehensive automated tests to ensure the quality and functionality of your web applications.

11.Write a simple program in JavaScript and perform testing using Selenium.

Let's create a simple JavaScript program that calculates the sum of two numbers and then perform testing using Selenium WebDriver in JavaScript.

Step 1: Create a JavaScript Program

Create a file named `calculator.js` and add the following JavaScript code:

```
function add(a, b) {
   return a + b;
}

console.log(add(2, 3)); // Output should be 5
```

This simple program defines a function `add` that takes two numbers as input and returns their sum. We then call this function with the numbers 2 and 3 and log the result to the console.

Step 2: Set up Selenium WebDriver in JavaScript

1. Install Selenium WebDriver for JavaScript using npm:

   ```
   npm install selenium-webdriver
   ```

2. Download the appropriate browser driver for the browser you want to automate testing with (e.g., ChromeDriver for Google Chrome).

Step 3: Write Selenium Test Script

Create a file named `test_calculator.js` and add the following JavaScript code to perform testing using Selenium WebDriver:

```
const { Builder, By, Key, until } = require('selenium-webdriver');

async function testAddition() {
   let driver = await new Builder().forBrowser('chrome').build();
```

```
    try {
        await driver.get('https://example.com'); // Replace with URL of your
application
        const a = await driver.findElement(By.id('a'));
        const b = await driver.findElement(By.id('b'));
        const result = await driver.findElement(By.id('result'));

        await a.sendKeys('2');
        await b.sendKeys('3');
        await a.sendKeys(Key.RETURN);

        await driver.wait(until.elementTextIs(result, '5'), 1000); // Wait until
result is calculated
        console.log('Test Passed!');
    } catch (e) {
        console.error('Test Failed!', e);
    } finally {
        await driver.quit();
    }
}

testAddition();
```

Step 4: Run Selenium Test Script

Save your test script to a file (e.g., `test_calculator.js`) and run it using Node.js:

```
node test_calculator.js
```

This script opens a browser, navigates to a webpage (you need to replace 'https://example.com' with the URL of your application), finds input elements with IDs 'a' and 'b', enters the numbers 2 and 3 into these inputs, and then verifies that the sum is calculated correctly.

Make sure to replace `'https://example.com'` with the URL of your application and adjust the element locators (`By.id`) according to the structure of your application.

12.Develop test cases for the above containerized application using selenium.

Let's develop test cases for the containerized Node.js application we created earlier using Selenium WebDriver in JavaScript.

Assuming our Node.js application serves a simple webpage with a form to input two numbers and displays the sum when a button is clicked, here are some test cases we can create:

1. **Test Case 1: Verify Page Title**
   - Open the webpage.
   - Verify that the page title is correct.
2. **Test Case 2: Verify Addition Functionality**
   - Input two numbers into the form.
   - Click the button to calculate the sum.
   - Verify that the displayed sum is correct.
3. **Test Case 3: Verify Error Handling**
   - Input invalid characters (e.g., letters) into the form.
   - Click the button to calculate the sum.
   - Verify that an error message is displayed.

Let's implement these test cases using Selenium WebDriver in JavaScript:

```
const { Builder, By, Key, until } = require('selenium-webdriver');

async function testPageTitle() {
    let driver = await new Builder().forBrowser('chrome').build();
    try {
        await driver.get('http://localhost:3000'); // Assuming the application is running locally
        const title = await driver.getTitle();
        console.log('Page Title:', title);
    } finally {
        await driver.quit();
```

```
    }
}

async function testAdditionFunctionality() {
    let driver = await new Builder().forBrowser('chrome').build();
    try {
        await driver.get('http://localhost:3000');
        const a = await driver.findElement(By.id('input-a'));
        const b = await driver.findElement(By.id('input-b'));
        const addButton = await driver.findElement(By.id('add-button'));

        await a.sendKeys('2');
        await b.sendKeys('3');
        await addButton.click();

        const result = await driver.findElement(By.id('result'));
        const sum = await result.getText();
        console.log('Sum:', sum);
    } finally {
        await driver.quit();
    }
}

async function testErrorHandling() {
    let driver = await new Builder().forBrowser('chrome').build();
    try {
        await driver.get('http://localhost:3000');
        const a = await driver.findElement(By.id('input-a'));
        const b = await driver.findElement(By.id('input-b'));
        const addButton = await driver.findElement(By.id('add-button'));

        await a.sendKeys('abc'); // Invalid input
        await b.sendKeys('def'); // Invalid input
```

```
        await addButton.click();

        const errorMessage = await driver.findElement(By.id('error-message'));
        const errorText = await errorMessage.getText();
        console.log('Error Message:', errorText);
    } finally {
        await driver.quit();
    }
}

async function runTests() {
    console.log('Running Test Cases:');
    console.log('====================');
    await testPageTitle();
    await testAdditionFunctionality();
    await testErrorHandling();
}

runTests();
```

In this script, we have three functions representing our test cases (`testPageTitle`, `testAdditionFunctionality`, `testErrorHandling`). Each function opens the webpage, interacts with the elements as required, and performs the necessary assertions. Finally, we have a `runTests` function that runs all the test cases sequentially. Make sure to replace `'http://localhost:3000'` with the URL of your application.