

## DevOps LAB Manual

### DevOps lab programs (Nalla Narasimha Reddy Engineering College)

**EXPERIMENT NO: 1. Write code for a simple user registration form for an event.**

**Aim: Write code for a simple user registration form for an event. DESCRIPTION:**

Here's an example of a simple user registration form using Flask and Docker in DevOps:

- Create a Docker file with the following content to create a Docker image for your Flask application:

```
FROM python:3.8 WORKDIR /app
```

```
COPY . .
```

```
RUN pip install --no-cache-dir -r requirements.txt EXPOSE 5000
```

```
CMD ["python", "app.py"]
```

- Create a requirements.txt file with the following content to list the dependencies of your Flask application: Flask==1.1.2
- Create a app.py file with the following code for a simple user registration form in Flask:

```
from flask import Flask, request, render_template
```

```
app = Flask(__name__)
```

```
@app.route('/register', methods=['GET', 'POST']) def register():
```

```
    if request.method == 'POST':  
        name = request.form['name']  
        email = request.form['email']  
        password = request.form['password']
```

```
    # Store the user data in a database or file
```

```
    return render_template('success.html')
```

```
return render_template('register.html')
```

```
if _name_ == '_main_':  
    app.run(host='0.0.0.0')
```

- Create a templates folder and add the following two files: register.html and success.html.

register.html

```
<form method="post">  
  
    <input type="text" name="name" placeholder="Name">  
  
    <input type="email" name="email" placeholder="Email">  
  
    <input type="password" name="password" placeholder="Password">  
  
    <input type="submit" value="Submit">
```

</form> success.html

```
<h2>Registration Successful</h2>
```

- Build the Docker image for your Flask application using the following command:  
docker build -t simple-flask-app .
- Run a Docker container from the image using the following command: docker run -p 5000:5000 simple-flask-app
- Open a web browser and access the registration form at <http://localhost:5000/register>.

This example demonstrates how to build a simple user registration form in Flask and run it in a Docker container in DevOps. Note that this code is only meant to illustrate the basic structure of a user registration form and does not include any security measures or proper error handling. Adding security measures such as password hashing and validation is highly recommended before using it in a production environment.

**EXPERIMENT NO: 2. Explore Git and GitHub commands Aim: Explore Git and GitHub commands**

## **DESCRIPTION:**

Git and GitHub are two of the most popular tools used for version control and collaboration in software development.

Here are some common Git and GitHub commands:

- Initializing a Git repository: `$ git init`
- Checking the status of your repository: `$ git status`
- Adding files to the stage: `$ git add <file-name>`
- Committing changes: `$ git commit -m "commit message"`
- Checking the commit history: `$ git log`
- Undoing changes: `$ git checkout <file-name>`
- Creating a new branch: `$ git branch <branch-name>`
- Switching to a different branch: `$ git checkout <branch-name>`
- Merging two branches: `$ git merge <branch-name>`
- Pushing changes to a remote repository: `$ git push origin <branch-name>`
- Cloning a repository from GitHub: `$ git clone <repository-url>`
- Creating a pull request on GitHub: Go to the repository on GitHub, select the branch we want to merge, and click the "New pull request" button.

These are just a few of the many Git and GitHub commands available. There are many other Git commands and functionalities that we can explore to suit our needs.

## **EXPERIMENT NO: 3. Practice Source code management on GitHub. Experiment with the source code written in Exercise 1**

**Aim:** Practice Source code management on GitHub. Experiment with the source code written in Exercise 1

### **Description:**

To practice source code management on GitHub, we can follow these steps:

- Create a GitHub account if we don't already have one.
- Create a new repository on GitHub.
- Clone the repository to your local machine: `$ git clone <repository- url>`
- Move to the repository directory: `$ cd <repository-name>`
  - Create a new file in the repository and add the source code written in exercise 1.
- Stage the changes: `$ git add <file-name>`
  - Commit the changes: `$ git commit -m "Added source code for a simple user registration form"`
- Push the changes to the remote repository: `$ git push origin master`
  - Verify that the changes are reflected in the repository on GitHub. These steps demonstrate how to use GitHub for source code management. We can use the same steps to manage any source code projects on GitHub. Additionally, we can also explore GitHub features such as pull requests, code review, and branch management to enhance your source code management workflow.

## **EXPERIMENT NO: 4. Jenkins installation and setup, explore the environment**

**Aim: Jenkins installation and setup, explore the environment**

# **DESCRIPTION**

Jenkins is a popular open-source tool for Continuous Integration and Continuous Deployment (CI/CD) in software development. Here are the steps to install and set up Jenkins:

Download and install Jenkins:

- Download the Jenkins package for your operating system from the Jenkins website.
- Follow the installation instructions for your operating system to install Jenkins.

Start the Jenkins service:

- On Windows, use the Windows Services Manager to start the Jenkins service.
  - On Linux, use the following command to start the Jenkins service:

`$ sudo service jenkins start` Access the Jenkins web interface:

- Open a web browser and navigate to `http://localhost:8080` to access the Jenkins web interface.
- If the Jenkins service is running, we will see the Jenkins login page. Initialize the Jenkins environment:
  - Follow the instructions on the Jenkins setup wizard to initialize the Jenkins environment.
  - This process involves installing recommended plugins, setting up security, and creating the first admin user.

Explore the Jenkins environment:

- Once the Jenkins environment is set up, we can explore the various features and functionalities available in the web interface.
- Jenkins has a rich user interface that provides access to features such as build history, build statistics, and system information.

These are the basic steps to install and set up Jenkins. Depending on your use case, we may need to customize your Jenkins environment further. For example, we may need to configure build agents, set up build pipelines, or integrate with other tools. However, these steps should give a good starting point for using Jenkins for CI/CD in your software development projects.

## **EXPERIMENT NO: 5. Demonstrate continuous integration and development using Jenkins.**

**Aim: Demonstrate continuous integration and development using Jenkins.**

# **DESCRIPTION**

Continuous Integration (CI) and Continuous Development (CD) are important practices in software development that can be achieved using Jenkins. Here's an example of how we can demonstrate CI/CD using Jenkins:

Create a simple Java application:

- Create a simple Java application that we want to integrate with Jenkins.
- The application should have some basic functionality, such as printing "Hello World" or performing simple calculations.

Commit the code to a Git repository:

- Create a Git repository for the application and commit the code to the repository.
- Make sure that the Git repository is accessible from the Jenkins server.

Create a Jenkins job:

- Log in to the Jenkins web interface and create a new job.
  - Configure the job to build the Java application from the Git repository.
  - Specify the build triggers, such as building after every commit to the repository.

Build the application:

- Trigger a build of the application using the Jenkins job.
- The build should compile the code, run any tests, and produce an executable jar file.

Monitor the build:

- Monitor the build progress in the Jenkins web interface.
- The build should show the build log, test results, and the status of the build.

Deploy the application:

- If the build is successful, configure the Jenkins job to deploy the application to a production environment.
- The deployment could be as simple as copying the jar file to a production server or using a more sophisticated deployment process, such as using a containerization technology like Docker.

Repeat the process:

- Repeat the process for subsequent changes to the application.
- Jenkins should automatically build and deploy the changes to the production environment.

This is a basic example of how we can use Jenkins to demonstrate CI/CD in software development. In a real-world scenario, we would likely have more complex requirements, such as multiple environments, different types of tests, and a more sophisticated deployment process. However, this example should give a good starting point for using Jenkins for CI/CD in your software development projects.



## EXPERIMENT NO: 6. Explore Docker commands for content management.

**AIM:** Explore Docker commands for content management.

# DESCRIPTION

Docker is a containerization technology that is widely used for managing application containers. Here are some commonly used Docker commands for content management:

- Docker run: Run a command in a new container.

For example: `$ docker run --name mycontainer -it ubuntu:16.04 /bin/bash`

This command runs a new container based on the Ubuntu 16.04 image and starts a shell session in the container.

- Docker start: Start one or more stopped containers. For example: `$ docker start mycontainer`

This command starts the container named "mycontainer".

- Docker stop: Stop one or more running containers. For example: `$ docker stop mycontainer`

This command stops the container named "mycontainer".

- Docker rm: Remove one or more containers. For example: `$ docker rm mycontainer`

This command removes the container named "mycontainer".

- Docker ps: List containers. For example:  
`$ docker ps`

This command lists all running containers.

- Docker images: List images. For example:  
`$ docker images`

This command lists all images stored locally on the host.

- Docker pull: Pull an image or a repository from a registry. For example: `$ docker pull ubuntu:16.04`

This command pulls the Ubuntu 16.04 image from the Docker Hub registry.

- Docker push: Push an image or a repository to a registry. For example: `$ docker push myimage`

This command pushes the image named "myimage" to the Docker Hub registry.

These are some of the basic Docker commands for managing containers and images. There are many other Docker commands and options that we can use for more advanced use cases, such as managing networks, volumes, and configuration. However, these commands should give a good starting point for using Docker for content management.

## EXPERIMENT NO.: 7. Develop a simple containerized application using Docker

**AIM: Develop a simple containerized application using Docker**

### DESCRIPTION

Here's an example of how we can develop a simple containerized application using Docker:

Choose an application:

- Choose a simple application that we want to containerize. For example, a Python script that prints "Hello World".

Write a Dockerfile:

- Create a file named "Dockerfile" in the same directory as the application.

In the Dockerfile, specify the base image, copy the application into the container, and specify the command to run the application. Here's an example Dockerfile for a Python script:

```
# Use the official Python image as the base image FROM
python:3.9
```

```
# Copy the Python script into the container COPY hello.py /app/
```

```
# Set the working directory to /app/ WORKDIR /app/
```

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

- Build the Docker image:

Run the following command to build the Docker image:

```
$ docker build -t myimage .
```

This command builds a new Docker image using the Dockerfile and tags the image with the name "myimage".

- Run the Docker container:

Run the following command to start a new container based on the image:

```
$ docker run --name mycontainer myimage
```

This command starts a new container named "mycontainer" based on the "myimage" image and runs the Python script inside the container.

- Verify the output:

Run the following command to verify the output of the container:

```
$ docker logs mycontainer
```

This command displays the logs of the container and should show the "Hello World" output.

This is a simple example of how we can use Docker to containerize an application. In a real-world scenario, we would likely have more complex requirements, such as running multiple containers, managing network connections, and persisting data. However, this example should give you a good starting point for using Docker to containerize your applications.

## **EXPERIMENT NO.: 8. Integrate Kubernetes and Docker**

### **AIM: Integrate Kubernetes and Docker**

## **DESCRIPTION:**

Kubernetes and Docker are both popular technologies for managing containers, but they are used for different purposes. Kubernetes is an orchestration platform that provides higher-level abstractions for managing containers, while Docker is a containerization technology that provides a lower-level runtime for containers.

To integrate Kubernetes and Docker, we need to use Docker to build and package your application as a container image, and then use Kubernetes to manage and orchestrate the containers.

Here's a high-level overview of the steps to integrate Kubernetes and Docker:

- Build a Docker image:

Use Docker to build a Docker image of your application. We can use a Dockerfile to specify the base image, copy the application into the container, and specify the command to run the application.

- Push the Docker image to a registry:

Push the Docker image to a container registry, such as Docker Hub or Google Container Registry, so that it can be easily accessed by Kubernetes. Deploy the Docker image to a Kubernetes cluster:

Use Kubernetes to deploy the Docker image to a cluster. This involves creating a deployment that specifies the number of replicas and the image to be used and creating a service that exposes the deployment to the network.

Monitor and manage the containers:

Use Kubernetes to monitor and manage the containers. This includes scaling the number of replicas, updating the image, and rolling out updates to the containers.

- Continuously integrate and deploy changes:

Use a continuous integration and deployment (CI/CD) pipeline to automatically build, push, and deploy changes to the Docker image and the Kubernetes cluster. This makes it easier to make updates to the application and ensures that the latest version is always running in the cluster.

By integrating Kubernetes and Docker, we can leverage the strengths of both technologies to manage containers in a scalable, reliable, and efficient manner.

## **EXPERIMENT NO.: 9. Automate the process of running containerized application developed in exercise 7 using Kubernetes**

**AIM: Automate the process of running containerized application developed in exercise 7 using Kubernetes**

# **DESCRIPTION**

To automate the process of running the containerized application developed in exercise 7 using Kubernetes, we can follow these steps:

- Create a Kubernetes cluster:

Create a Kubernetes cluster using a cloud provider, such as Google Cloud or Amazon Web Services, or using a local installation of Minikube.

- Push the Docker image to a registry:

Push the Docker image of your application to a container registry, such as Docker Hub or Google Container Registry.

- Create a deployment:

Create a deployment in Kubernetes that specifies the number of replicas and the Docker image to use. Here's an example of a deployment YAML file: apiVersion: apps/v1

kind: Deployment metadata:

name: myapp spec:

replicas: 3 selector:

matchLabels:

app: myapp template:

metadata:

labels:

app: myapp spec:

containers:

- name: myapp image: myimage

ports:

- containerPort: 80

- Create a service:

Create a service in Kubernetes that exposes the deployment to the network. Here's an example of a service YAML file:

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

```
metadata:
```

```
  name: myapp-service spec:
```

```
  selector:
```

```
    app: myapp ports:
```

```
      - name: http port: 80
```

```
  targetPort: 80 type: ClusterIP
```

- Apply the deployment and service to the cluster:

Apply the deployment and service to the cluster using the kubectl command-line tool.

For example:

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

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

- Verify the deployment:

Verify the deployment by checking the status of the pods and the service. For example:

```
$ kubectl get pods
```

```
$ kubectl get services
```

This is a basic example of how to automate the process of running a containerized application using Kubernetes. In a real-world scenario, we would likely have more complex requirements, such as managing persistent data, scaling, and rolling updates, but this example should give a good starting point for using Kubernetes to manage your containers.



## EXPERIMENT NO.: 10. Install and Explore Selenium for automated testing

**AIM:** Install and Explore Selenium for automated testing

# DESCRIPTION:

To install and explore Selenium for automated testing, we can follow these steps:

Install Java Development Kit (JDK):

- Selenium is written in Java, so you'll need to install JDK in order to run it. We can download and install JDK from the official Oracle website.
- Install the Selenium WebDriver:
  - we can download the latest version of the Selenium WebDriver from the Selenium website. You'll also need to download the appropriate driver for your web browser of choice (e.g. Chrome Driver for Google Chrome).

Install an Integrated Development Environment (IDE):

- To write and run Selenium tests, you'll need an IDE. Some popular choices include Eclipse, IntelliJ IDEA, and Visual Studio Code.
- Write a simple test:
  - Once we have your IDE set up, we can write a simple test using the Selenium WebDriver. Here's an example in Java:

```
import org.openqa.selenium.WebDriver;
```

```
import org.openqa.selenium.chrome.ChromeDriver;
```

```
public class Main {  
    public static void main(String[] args) { System.setProperty("webdriver.chrome.driver",  
        "path/to/chromedriver"); WebDriver driver = new ChromeDriver();  
        driver.get("https://www.google.com"); System.out.println(driver.getTitle());  
  
        driver.quit();  
    }  
}
```

```
}  
  
}
```

- Run the test:

Run the test using your IDE or from the command line using the following command:

```
$ javac Main.java
```

```
$ java Main
```

This is a basic example of how to get started with Selenium for automated testing. In a real-world scenario, we would likely write more complex tests and organize our code into test suites and test cases, but this example should give a good starting point for exploring Selenium.

## EXPERIMENT NO.: 11. Write a simple program in JavaScript and perform testing using Selenium

**AIM:** Write a simple program in JavaScript and perform testing using Selenium

### PROGRAM:

- Simple JavaScript program that we can test using Selenium

```
<!DOCTYPE html>
<html>
<head>
  <title>Simple JavaScript Program</title>
</head>
<body>
  <p id="output">0</p>
  <button id="increment-button">Increment</button>
  <script>
    const output = document.getElementById("output");
    const incrementButton =
document.getElementById("increment-button");

    let count = 0;

    incrementButton.addEventListener("click", function() {
      count += 1;
      output.innerHTML = count;
    });
  </script>
</body>
</html>
```

- Write a test case for this program using Selenium

```
import org.openqa.selenium.By;
```

```
import org.openqa.selenium.WebDriver;
```

```
import org.openqa.selenium.chrome.ChromeDriver;
```

```
import org.junit.After;
```

```
import org.junit.Before;
```

```
import org.junit.Test;
```

```
public class Main {
```

```
    private WebDriver driver;
```

```
    @Before
```

```
    public void setUp() {
```

```
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
```

```
        driver = new ChromeDriver();
```

```
    }
```

```
    @Test
```

```
    public void testIncrementButton() {
```

```
        driver.get("file:///path/to/program.html");
```

```
        driver.findElement(By.id("increment-button")).click();
```

```
        String result = driver.findElement(By.id("output")).getText();
```

```
        assert result.equals("1");
```

```
    }
```

```
    @After
```

```
    public void tearDown() {
```

```
        driver.quit();
```

```
}  
  
}
```

We can run the test case using the following command:

```
$ javac Main.java
```

```
$ java Main
```

The output of the test case should be:

```
.
```

```
Time: 0.189
```

```
OK (1 test)
```

This output indicates that the test case passed, and the increment button was successfully clicked, causing the output to be incremented by 1.

**EXPERIMENT NO:12. Develop test cases for the above containerized application using selenium**

**AIM: Develop test cases for the above containerized application using selenium**

## PROGRAM:

Here is an example of how we could write test cases for the containerized application using Selenium

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class Main {

    private WebDriver driver;

    @Before

    public void setUp() {

        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

        driver = new ChromeDriver();

    }

    @Test
    public void testHomePageLoads() {
        driver.get("http://localhost:8080");
        String title = driver.getTitle();

        assert title.equals("My Containerized Application");

    }

    @Test
    public void testSubmitForm() {
        driver.get("http://localhost:8080");
        driver.findElement(By.name("name")).sendKeys("John Doe");

        driver.findElement(By.name("email")).sendKeys("john.doe@example.com");

        driver.findElement(By.name("submit")).click();
    }
}
```

```
String result = driver.findElement(By.id("result")).getText(); assert
result.equals("Form submitted successfully!");

}
```

@After

```
public void tearDown() {
    driver.quit();
}

}
```

We can run the test cases using the following command:

```
$ javac Main.java
```

```
$ java Main
```

The output of the test cases should be:

```
..
```

Time: 1.135

OK (2 tests)

This output indicates that both test cases passed, and the containerized application is functioning as expected.