B.Tech (CSE). III Year – I Semester

DevOps Lab Manual

Prepared By:
L Suresh
Asst.Prof.(CSE)
CJITS, Jangaon

LIST OF EXPERIMENTS

S.No.	Name of the Experiment
1.	Write code for a simple user registration formfor an event.
2.	Explore Git and GitHub commands
3.	Practice Source code management on GitHub. Experiment with the source code written in exercise 1
4.	Jenkins installation and setup, explore the environment
5.	Demonstrate continuous integration and development using Jenkins.
6.	Explore Docker commands for content management.
7.	Develop a simple containerized application using Docker
8.	Integrate Kubernetes and Docker
9.	Automate the process of running containerized application developed in exercise 7 using Kubernetes
10.	Install and Explore Selenium for automated testing
11.	Write a simple program in JavaScript and perform testing using Selenium
12.	Develop test cases for the above containerized application using selenium.

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:

Program Structure:

```
exp1/
├── app.py
├── templates/
|   ├── register.html
|   └── success.html
├── requirements.txt
└── Dockerfile
```

1. app.py Code (Flask Back End):

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

app = Flask(__name__)

# Temporary storage (replace with a database in production)
users = []

@app.route('/')
def home():
    return redirect(url_for('register'))

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        email = request.form['email']
        password = request.form['password']

        users.append({'username': username, 'email': email})
        return redirect(url_for('success'))

    return render_template('register.html')

@app.route('/success')
def success():
    return render_template('success.html')
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

2. templates/register.html (Registration Form):

```
<html>
<head>
    <title>Register</title>
</head>
<body>
    <h1>Register</h1>
    <form method="POST">
        <input type="text" name="username" placeholder="Username"
required><br>
        <input type="email" name="email" placeholder="Email"
required><br>
        <input type="password" name="password"
placeholder="Password" required><br>
        <button type="submit">Register</button>
    </form>
</body>
</html>
```

3. templates/success.html (Success Page) :

```
<html>
<head>
    <title>Success</title>
</head>
<body>
    <h1>Registration Successful!</h1>
    <p>Thank you for registering.</p>
</body>
</html>
```

4. requirements.txt (Dependencies) :

```
Flask==2.3.2
```

5. Dockerfile (Containerization) :

```
FROM python:3.9-slim
WORKDIR /app
```

```
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

How to Run It

1. **Build and Run with Docker**:

```
docker build -t exp1 .
docker run -p 5000:5000 exp1
```

2. **Access the App**:
   Open http://localhost:5000/register in your browser.

Explanation of Program

Flask is a lightweight **Python web framework** used for building web applications and APIs quickly and efficiently. It's known for its simplicity, flexibility, and minimalistic design, making it ideal for small to medium projects and microservices.

App.py:

1. Import Statements

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

- Flask: Main Flask class to create the application instance
- render_template: Renders HTML templates from the templates/ folder
- request: Handles HTTP request data (form submissions)
- redirect: Redirects to another route
- url_for: Generates URLs for Flask routes

2. Flask Application Setup

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

- Creates a Flask application instance

- __name__ tells Flask where to look for templates/static files.

3. Temporary Storage

```
users = []
```

- In-memory list to store registered users (**For demo only** - use a database in production)

4. Home Route

```
@app.route('/')
def home():
    return redirect(url_for('register'))
```

- @app.route('/'): Maps the root URL (/) to this function
- redirect(url_for('register')): Redirects to the /register route

5. Registration Route (GET/POST)

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

- methods=['GET', 'POST']: Accepts both GET (page load) and POST (form submission) requests

  ✓ ***POST Request Handling:***

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

request.form: Accesses form data submitted via POST
Extracts username, email, and password from the form

```
users.append({'username': username, 'email': email})
```

- Stores user data in the users list (without password for demo safety)

```
return redirect(url_for('success'))
```

- Redirects to the success page after registration

  - ✓ ***GET Request Handling***

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

- Renders the registration form template for GET requests

6. Success Route

```
@app.route('/success')
def success():
    return render_template('success.html')
```

- Displays a success page after registration

7. Application Entry Point

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

- host='0.0.0.0': Makes the server accessible outside the local machine (required for Docker)
- debug=True: Enables auto-reloader and debugger (disable in production!)

EXPERIMENT NO: 2. Explore Git and GitHub commands

Aim: Explore Git and GitHub commands.

Description:

1. What is Git?

Git is a **distributed version control system (DVCS)** that:

- Tracks changes in your code over time.
- Allows branching/merging for parallel development.
- Works **offline** (all history is stored locally).

<u>Key Git Concepts:</u>

- **Repository (Repo)**: A folder where Git tracks files.
- **Commit**: A snapshot of changes at a point in time.
- **Branch**: Isolated line of development (e.g., `main`, `feature/login`).
- **Merge**: Combines changes from different branches.

2. What is GitHub?

GitHub is a **cloud-based platform** that:

- Hosts Git repositories remotely.
- Enables collaboration via **pull requests**, **issues**, and **projects**.
- Provides additional tools like **GitHub Actions (CI/CD)** and **Pages**.

- ✓ **Install Git: from https://www.git-scm.com (for Windows OS)**
- ✓ **Create GitHub account : https://www.github.com**

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 you 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 you can explore to suit your 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, you can follow these steps:

- ✓ Create a GitHub account if you 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.
- ✓ You can use the same steps to manage any source code projects on GitHub. Additionally, you can also explore GitHub features such as pull requests, code review, and branch management to enhance your source code management workflow.

1. Initialize Git in Your Local Folder

```
$ cd path/to/your-folder

# Initialize Git repository
$ git init
```

2. Stage and Commit Files

```
# Stage all files (or use `git add <filename>` for specific files)
$ git add .
# Commit with a message
$ git commit -m "Initial commit"
```

3. Create a New Repository on GitHub

Go to github.com/new
Enter a repository name (e.g., `exp1`)
Do not initialize with README/.gitignore (keep it empty)
Click "Create repository"

4. Link Local Repository to GitHub

```
# Copy the remote repository URL (HTTPS or SSH) from GitHub
$ git remote add origin https://github.com/your-username/your-repo-name.git
```

5. Push to GitHub

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

6. Verify on GitHub

Refresh your GitHub repository page. Your files should now appear!

Summary

1. `git init` → `git add .` → `git commit`
2. Connect to GitHub with `git remote add origin`
3. `git push -u origin main`

Your local folder is now on GitHub!

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

Aim: Jenkins installation and setup, explore the environment
DESCRIPTION:
Jenkins: The Ultimate DevOps Automation Tool

Jenkins is an **open-source automation server** used for **CI/CD (Continuous Integration & Continuous Delivery)**. It automates building, testing, and deploying software, making DevOps workflows faster and more reliable.

Download and install Jenkins:

- Download the Jenkins package for your operating system from the
  - Jenkins website. -- > https://www.jenkins.io/download/
- 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.
- 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, you 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, you 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, you may need to customize your Jenkins environment further. For example, you may need to configure build agents, set up build pipelines, or integrate with other tools. However, these steps should give you 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 you can demonstrate CI/CD using Jenkins:

Create a simple Java application that you 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.

Create a Simple Java Application

Let's create a basic Java app that prints "Hello, World!" and performs a simple addition.

1. Open Visual Studio Code:
   Press Ctrl + Shift + P

- Select Java: Create Java Project
- Select Folder to create Java Project.
- Type Project Name: SimpleJavaApp

Select Left side **src** folder change name App.java to Main.Java

Remove code and type below code:

Main.java:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        int a = 5, b = 7;
        int sum = a + b;
        System.out.println("Sum: " + sum);
    }
}
```

build.bat:

```
echo Building Java App...
mkdir out 2>nul
javac -d out src\Main.java

echo Running Java App...
cd out
```

```
java Main
echo Build and run completed.
```

- **Debug the Project**

- Open the https://github.com/ website and login your account and create a new repository.
- Goto your project folder in local system.
  D:\Project_Folder\

- **Run cmd in addressbar.**
  Type below commands:
  1. git init
  2. git add .
  3. git config --global user.name "cjits25"
  4. git config --global user.email "cjitsjn@gmail.com"
  5. git commit –m "Intiated Project commited"
  6. git push –u origin master
- Open the Chrome Browser type the URL : http://localhost:8080

Set Up Jenkins Job

- **Step-by-Step in Jenkins UI:**

1. **Login to Jenkins**
2. Click **"New Item"**
3. Enter item name: SimpleJavaApp-Build
4. Choose **"Freestyle project"**
5. Click **OK**

- **Configure Source Code Management:**

- Select **Git**
- Enter your Git repository URL (e.g., https://github.com/your-username/SimpleJavaApp.git)

- **Configure Build:**
  - Click **"Add build step" → "Execute Windows batch command"**.

- **Archive .class Files as Build Artifacts**

  Steps:

  1. Go to your Jenkins job → **Configure**
  2. Scroll to **Post-build Actions**
  3. Click **Add post-build action** → **Archive the artifacts**
  4. In **Files to archive**, enter:

- Click **Save**.
- Click **Build Now**.

Output in **Console Output** of the build log.

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:

Goto the run : type the command 'cmd'.

Docker Basics on Windows :

1. Check Docker is Running:

   C:\> docker version

   Displays the **installed Docker version** on your system (client and server).

   C:\> docker info

   Shows detailed information like running containers, images, volumes, and Docker configuration.

Image Management Commands:

2. List all images:

   C:\> docker images

Lists all the **Docker images** downloaded on your system, including:

- Repository name
- Tag (version)
- Image ID
- Creation time
- Size

C:\> docker pull <image-name>

Downloads a Docker image from **Docker Hub** (or another registry) to your local machine.

- Example: c:\> docker pull nginx pulls the latest NGINX image.

C:\> docker rmi <image-id or image-name>

Removes a Docker image from your local system to free up space.

- Example: docker rmi nginx or docker rmi abc123456789

Container Management Commands:

C:\>docker ps -a

Lists **all containers**, including:

- Running containers
- Stopped or exited containers

Use docker ps without -a to see only running containers.

C:\>docker start <container-id or name>

Starts a **stopped container**.

- You must have created the container earlier.
- Example: docker start myweb

C:\>docker stop <container-id or name>

Gracefully stops a **running container**.

C:\>docker run -it --name mycontainer ubuntu

Runs a new **Ubuntu container** interactively:

- -i: keep STDIN open
- -t: allocate a terminal
- --name: gives the container a name
- Example: you get a terminal shell inside Ubuntu.

C:\>docker run -d --name myweb nginx

Runs a container in **detached mode (background)**.

- -d: detached mode
- Container runs independently until stopped

C:\>docker rm <container-id or name>

Removes a **stopped container**.

- You must stop the container first.
- Example: docker rm myweb

::VIVA::

What is Docker?

Docker is an **open-source platform** used to **build, ship, and run applications inside containers**.

- It allows you to package an application and its dependencies into a **single lightweight unit** called a **container**.
- Containers run the same regardless of the underlying OS or hardware.

What is a Container?

A **container** is a lightweight, standalone executable package that includes:

- The application code
- Runtime (e.g., Java, Python, Node)
- Libraries and dependencies
- Configuration files

Think of it as a **mini virtual environment**, but faster and lighter than virtual machines.

How Docker Works:

Docker uses:

- **Docker Engine**: Runs and manages containers.
- **Dockerfile**: Script to define how to build a container image.
- **Docker Hub**: A public registry where you can find and share container images.
- **Docker CLI**: Command-line tool to interact with Docker.

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 you can develop a simple containerized application using Docker:

Choose an application:

- Choose a simple application that you 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:

hello.py:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Docker World!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

requirements.txt

```
flask
```

Dockerfile:

```
# Use an official Python image
FROM python:3.11-slim

# Set working directory
WORKDIR /app
```

```
# Copy app files
COPY requirements.txt requirements.txt
COPY app.py app.py

# Install dependencies
RUN pip install -r requirements.txt

# Expose the port Flask runs on
EXPOSE 5000

# Command to run the app
CMD ["python", "app.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 you can use Docker to containerize an application. In a real-world scenario, you 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, you 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. You 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, you can leverage the strengths ofboth 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, you can follow these steps:

- **<u>Create a Kubernetes cluster:</u>**

Create a Kubernetes cluster using a cloud provider, such as Google Cloud or

Amazon Web Services, or using a local installation of Minikube.

- **<u>Push the Docker image to a registry:</u>**

Push the Docker image of your application to a container registry, such as

Docker Hub or Google Container Registry.

- **<u>Create a deployment:</u>**

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: myappspec:
 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: v1kind: 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, you would likely have more complex requirements, such as managing persistent data, scaling, and rolling updates, but this example should give you a good starting point for exploring Selenium.

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, you 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. You can download and install JDK from the official Oracle website.
- Install the Selenium WebDriver:
- You 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 you have your IDE set up, you can write a simple test using theSelenium 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, you would likely write more complex tests and organize your code into test suites and test cases, but this example should give you 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 usingSelenium

PROGRAM:

- Simple JavaScript program that you 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>
```

12. Develop test cases for the above 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 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");
 }
```