J.N.T.U.H. UNIVERSITY COLLEGE OF ENGINEERING, SCIENCE & TECHNOLOGY
HYDERABAD

KUKATPALLY, HYDERABAD – 500 085

Certificate

Certified that this is the bonafide record of the practical work done during

the academic year _____ by

Name _____ Roll Number

_____ Class _____

in the Laboratory of _____

of the Department of _____

Signature of the Staff Member

Signature of the Head of the Department

Date of Examination_____

Signature of the Examiner/s

Internal Examiner

External Examiner

List of Experiments

Experiment 1: Write Code for a Simple User Registration Form for an Event

Objective

To design and implement a simple user registration form for an event using HTML, CSS, and JavaScript for validation.

Software / Tools Required

- Any text editor (VS Code / Sublime / Notepad++)
- Web browser (Chrome / Edge / Firefox)

Theory

A user registration form collects basic user details such as name, email, phone number, gender, selected event, and address.
HTML defines the structure, CSS improves the visual appearance, and JavaScript provides form validation before submission.

Program Code (Event_registration.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Event Registration</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #f4f4f4;
    }

    .form-container {
      background-color: #fff;
      padding: 20px;
      border: 1px solid #ccc;
      width: 400px;
      margin: 50px auto;
    }
```

```
    h1 {
      text-align: center;
    }

    .form-row {
      margin-bottom: 15px;
    }

    .form-row label {
      display: block;
      margin-bottom: 5px;
    }

    input[type="text"],
    input[type="email"],
    input[type="tel"],
    select,
    textarea {
      width: 100%;
      padding: 8px;
      border: 1px solid #ccc;
      box-sizing: border-box;
    }

    .buttons {
      text-align: right;
      margin-top: 20px;
    }

    input[type="submit"],
    input[type="reset"] {
      padding: 8px 12px;
      border: 1px solid #999;
      cursor: pointer;
      background-color: lightgreen;
    }
  </style>
</head>

<body>

  <div class="form-container">
    <h1>Event Registration</h1>
```

```
    <form name="registrationForm" onsubmit="return
validateForm()">
      <div class="form-row">
        <label for="fullname">Full Name:</label>
        <input type="text" name="fullname" id="fullname"
required minlength="3">
      </div>

      <div class="form-row">
        <label for="email">Email ID:</label>
        <input type="email" name="email" id="email" required>
      </div>

      <div class="form-row">
        <label for="phno">Phone Number:</label>
        <input type="tel" name="phno" id="phno" required
pattern="[0-9]{10}" title="Please enter a 10-digit phone
number">
      </div>

      <div class="form-row">
        <label>Gender:</label>
        <input type="radio" name="gender" value="Male"
required> Male
        <input type="radio" name="gender" value="Female"
required> Female
      </div>

      <div class="form-row">
        <label for="event">Select Event:</label>
        <select id="event" name="event" required>
          <option value="">-- Please choose --</option>
          <option value="Workshop">Workshop</option>
          <option value="Seminar">Seminar</option>
          <option value="Networking">Networking</option>
        </select>
      </div>

      <div class="form-row">
        <label for="address">Address:</label>
        <textarea name="address" id="address" required></
textarea>
      </div>

      <div class="buttons">
```

```
      <input type="reset" value="Reset">
      <input type="submit" value="Submit">
    </div>
  </form>
</div>

<script>
  function validateForm() {
    const form = document.forms["registrationForm"];

    if (form.fullname.value.trim() === "") {
      alert("Please enter your Full Name.");
      form.fullname.focus();
      return false;
    }

    const email = form.email.value;
    if (email.trim() === "" || email.indexOf("@") === -1) {
      alert("Please enter a valid Email ID.");
      form.email.focus();
      return false;
    }

    const phno = form.phno.value;
    if (isNaN(phno) || phno.trim().length !== 10) {
      alert("Please enter a valid 10-digit Phone Number.");
      form.phno.focus();
      return false;
    }

    if (form.gender.value === "") {
      alert("Please select your Gender.");
      return false;
    }

    if (form.event.value === "") {
      alert("Please select an Event.");
      form.event.focus();
      return false;
    }

    if (form.address.value.trim() === "") {
      alert("Please enter your Address.");
      form.address.focus();
      return false;
```

```
      }

      alert("Registration Successful!");
      return true;
    }
  </script>
</body>
</html>
```

Output

A webpage displaying a clean event registration form with client-side validation.
When all fields are valid → shows **"Registration Successful!"** alert.

Result

Successfully created and validated an event registration form using HTML, CSS, and JavaScript.

Experiment 2: Explore Git and GitHub Commands

Objective

To explore version control operations using Git and GitHub for efficient source code management and collaboration.

Software / Tools Required

- Git (installed locally)
- GitHub account
- Command-line / Terminal

Theory

Git is a distributed version control system used for tracking changes in source code.
GitHub is a cloud-based hosting platform for managing Git repositories and collaborating with others.

Common Git Commands

Command	Description
`git init`	Initialize a local repository
`git clone <url>`	Clone a remote repository
`git status`	Show current repo status
`git add <file>`	Stage file for commit
`git commit -m "message"`	Commit changes
`git log`	View commit history
`git remote add origin <url>`	Link local repo to GitHub
`git push origin main`	Push commits to GitHub
`git pull origin main`	Pull updates from GitHub
`git branch`	List branches
`git checkout -b <branch>`	Create and switch branch
`git merge <branch>`	Merge a branch into current
`git diff`	Show differences between commits

Example Workflow

```
# Step 1: Initialize repo
git init

# Step 2: Add files
git add Event_registration.html

# Step 3: Commit
git commit -m "Initial commit - Event registration form"

# Step 4: Create remote repo on GitHub
git remote add origin https://github.com/<username>/event-
registration.git

# Step 5: Push to GitHub
git branch -M main
git push -u origin main
```

Result

Explored Git and GitHub commands for initializing, committing, pushing, and synchronizing repositories.

Experiment 3: Practice Source Code Management on GitHub

Objective

To practice source code versioning, synchronization, and collaboration using GitHub for the Event Registration form project.

Procedure

1. **Create a Local Project**
    - Save `Event_registration.html` in a folder named `event-registration`.
2. **Initialize Git Repository** — `git init`
3. **Add File and Commit**

```
git add Event_registration.html

git commit -m "Added event registration form"
```

4. **Create Repository on GitHub**
    - Go to GitHub → New Repository → Name: `event-registration`
5. **Link Local Repo to Remote**

```
git remote add origin https://github.com/<username>/
event-registration.git

git branch -M main

git push -u origin main
```

6. **Modify Code (Example)**
    - Add a new field, e.g., "Age".
    - Save file, then:

```
git add .
```

    - `git commit -m "Added age field to form"`
    - `git push`

13. **View Commit History**

```
git log --oneline
```

14. **Collaborate**
    - Invite others via GitHub Collaborators.
    - Test pull requests or forks.

Result

Successfully practiced version control and collaboration workflows on GitHub using the event registration form code.

Conclusion

Git and GitHub provide powerful tools for managing, tracking, and sharing source code changes. They are essential for teamwork, CI/CD pipelines, and modern DevOps practices.

Experiment 4: Jenkins Installation and Setup (macOS / Windows / Linux)

Objective

To install Jenkins on macOS, Windows, and Linux systems, explore its environment, and verify successful setup.

Software / Tools Required

- Java (JDK 11 or higher)
- Jenkins (LTS version)
- Git
- Web Browser

🐧 A. Installation on Linux (Ubuntu / Debian)

Step 1 — Install Java

```
sudo apt update
sudo apt install openjdk-17-jdk -y
java -version
```

Step 2 — Add Jenkins Repository and Install

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key |
sudo tee \
  /usr/share/keyrings/jenkins-keyring.asc > /dev/null


echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]
\
  https://pkg.jenkins.io/debian binary/ | sudo tee \
  /etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt update

sudo apt install jenkins -y
```

Step 3 — Start and Enable Jenkins

```
sudo systemctl enable jenkins
sudo systemctl start jenkins
sudo systemctl status jenkins
```

Step 4 — Access Jenkins

Open your browser and go to:

```
http://localhost:8080
```

Retrieve the admin password:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

Paste the password, install suggested plugins, and complete setup.

🍎 B. Installation on macOS (Intel / Apple Silicon)

Step 1 — Install Homebrew (if not already installed)

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/HEAD/install.sh)"
```

Step 2 — Install Java and Jenkins

```
brew install openjdk@17
brew install jenkins-lts
```

After installation, link Java if necessary:

```
sudo ln -sfn $(brew --prefix openjdk@17)/libexec/

openjdk.jdk /Library/Java/JavaVirtualMachines/openjdk-17.jdk
```

Step 3 — Start Jenkins

```
brew services start jenkins-lts
```

Step 4 — Access Jenkins

Open in your browser:

```
http://localhost:8080
```
Get the admin password:

```
cat /Users/Shared/Jenkins/Home/secrets/initialAdminPassword
```
Paste it into Jenkins → Install suggested plugins → Create admin user.

Optional Commands

To stop Jenkins:

```
brew services stop jenkins-lts
```

To restart:

```
brew services restart jenkins-lts
```

C. Installation on Windows (10 / 11)

Step 1 — Install Java

Download and install JDK 17 from the official Oracle or OpenJDK site.
Then verify in Command Prompt:

```
java -version
```

Step 2 — Install Jenkins

1. Go to: https://www.jenkins.io/download/
2. Download the **Windows .msi** installer.
3. Run the installer:
    - Select **"Run service as Local System"**
    - Default port: **8080**
    - Jenkins home directory: `C:\Program Files\Jenkins`
4. When setup completes, Jenkins will start as a Windows service.

Step 3 — Access Jenkins

Open:

```
http://localhost:8080
```

Unlock Jenkins using the password from:

```
C:\Program Files\Jenkins\secrets\initialAdminPassword
```

Then follow on-screen instructions → Install suggested plugins → Create admin user.

Step 4 — Verify Jenkins Service

Open **Services** (`services.msc`) → ensure **Jenkins** is *Running*.

To restart:

```
net stop jenkins
net start jenkins
```

Exploring the Jenkins Environment (Common Across All OS)

Feature	Description
Dashboard	Displays jobs and build history
Manage Jenkins	System configuration, security, plugin management
Build Executor Status	Shows available build agents
Credentials	Securely store Git/Docker credentials
New Item	Create Freestyle or Pipeline projects

Jenkins Job Creation

Here Jenkins clones the GitHub Repository "https://github.com/bharath-akurathi/devops-lab.git" and opens a local HTML file `Event-registration.html` for previewing.

Purpose:

To preview or test static HTML web pages automatically after every build.

Steps

1. **Create a New Item**

- From Jenkins Dashboard → **New Item**
- Choose **Freestyle Project**
- Name it: `html-viewing-github`
- Click **OK**

2. **Configure Source Code Management**
   - Choose **Git**
   - **Repository URL:** `https://github.com/bharath-akurathi/devops-lab.git`
   - **Branch:** `*/main`
3. **Build Steps**
   - Add **Execute Shell Command**
   - Use the appropriate command based on your OS:
     macOS — `open ./Event-registration.html`

     Linux — xdg-open ./Event-registration.html

     Windows (Git Bash / PowerShell) — start ./Event-registration.html
4. **Post-Build Actions**
   - Add **Publish HTML Reports** (install the *HTML Publisher Plugin* if not already available).
   - **HTML directory:** `1`
   - **Index page:** `Event-registration.html`
   - **Report title:** `Event Registration Page`

What This Job Demonstrates

- **Git Integration** → Jenkins automatically clones your GitHub repository.
- **Shell Execution** → Jenkins runs OS-specific commands to open or preview files.
- **HTML Reporting** → Jenkins can display HTML content directly within the job's results page.

Observation

- Jenkins environment explored successfully.
- Git integration and HTML publishing verified.
- The static HTML page was opened and previewed successfully.

Result

Jenkins was successfully installed and configured on **macOS, Windows, and Linux**.
The Jenkins environment and dashboard were explored successfully.

Experiment 5: Demonstrate Continuous Integration and Development using Jenkins

Objective

To demonstrate Continuous Integration (CI) and Continuous Deployment (CD) using Jenkins with a sample GitHub project.

Software / Tools Required

- Jenkins (running instance)
- Git & GitHub repository
- Web browser

Theory

In CI/CD:

- **Continuous Integration (CI)** — Developers commit code to GitHub → Jenkins automatically builds and tests the project.
- **Continuous Deployment (CD)** — After successful build, Jenkins automatically deploys or runs the application.

This reduces manual effort and ensures quick, reliable delivery.

Procedure

Step 1 — Start Jenkins

Ensure Jenkins is running and accessible at:

```
http://localhost:8080
```

Step 2 — Create a New Pipeline Job

- Click **"New Item"** → enter name:
  `devops_lab_ci_cd`
- Choose **Pipeline** → click **OK**

Step 3 — Configure Git Repository

In **Pipeline** section:

- Definition → *Pipeline script from SCM*
- SCM → **Git**
- Repository URL → `https://github.com/<your-username>/devops_lab.git`
- Branch Specifier → `*/main`

Step 4 — Add Jenkinsfile to the Repository

Create a new file in your GitHub repo named **Jenkinsfile**:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh '''
                    DEPLOY="$HOME/devops_lab_site"
                    mkdir -p "$DEPLOY"
                    cp -f website/Event_registration.html
"$DEPLOY"/Event_registration.html
                '''
            }
        }

        stage('Serve') {
            steps {
                sh '''
                    DEPLOY="$HOME/devops_lab_site"
                    FILE="$DEPLOY/Event_registration.html"
                    echo "File deployed to: $FILE"
                '''
            }
        }
    }
}
```

(If you're on Windows Jenkins, replace `sh` with `bat` and adjust paths accordingly.)

Step 5 — Save and Build

- Click **Save**
- Run the job using **Build Now**

Step 6 — View Build Output

Check the **Console Output**:

- Jenkins pulls the code from GitHub
- Runs the `Build` stage → copies HTML file to the deployment folder
- Runs the `Serve` stage → confirms file deployment

You can manually open the deployed file:

```
~/devops_lab_site/Event_registration.html
```

Observation

✅ Jenkins automatically pulled the source code from GitHub
✅ Built and deployed the HTML file
✅ Demonstrated automation of build + deploy pipeline

Result

Successfully implemented a **CI/CD pipeline using Jenkins**, which fetched code from GitHub, built the project, and deployed the output automatically.

Conclusion

Jenkins enables full automation of software development workflows — from source integration (CI) to deployment (CD).
It ensures faster delivery, consistency, and easy integration with GitHub and Docker/Kubernetes.

Experiment 6: Explore Docker Commands for Content Management

Objective

To explore various Docker commands used for managing images, containers, and data persistence using volumes and bind mounts.

Software / Tools Required

- Docker Engine (Desktop or CLI)
- Linux / macOS / Windows terminal

Theory

Docker provides a containerized environment that packages applications with all dependencies. To effectively use Docker, we need to manage images (application blueprints), containers (running instances), and persistent data using volumes.

This experiment explores key Docker commands for:

- **Image management** (pull, inspect, tag, save/load)
- **Container management** (create, copy files, commit, export/import)
- **Persistence** (volumes & bind mounts)

Procedure

A. Image Management

1. **Pull an image**
```
docker pull alpine:3.19
```

2. **List images**
```
docker images
```

3. **Inspect image metadata**
```
docker inspect alpine:3.19
```

4. **View layer history**
```
docker history alpine:3.19
```

5. **Tag and remove image**

```
docker tag alpine:3.19 alpine:lab

docker rmi alpine:lab
```

6. **Save and load image**

```
docker save -o alpine.tar alpine:3.19

docker load -i alpine.tar
```

B. Container Content Management

1. **Run an interactive container**

```
docker run -it --name labc alpine:3.19 sh
```

2. **Inside the container**

```
echo "hello from container" > /msg.txt

mkdir /data && echo "42" > /data/number.txt

exit
```

3. **Copy files to/from container**

```
docker cp labc:/msg.txt ./msg.txt

docker cp ./msg.txt labc:/data/
```

4. **Inspect container and logs**

```
docker inspect labc

docker logs labc
```

5. **Snapshot and export container**

```
docker commit labc alpine:with-data

docker export labc -o rootfs.tar

docker import rootfs.tar alpine:imported
```

6. **Remove container**

```
docker stop labc && docker rm labc
```

C. Persistence with Volumes and Bind Mounts

1. **Using Docker Volumes**

```
docker volume create labvol

docker run -it --name volc -v labvol:/appdata
alpine:3.19 sh

echo "persistent file" > /appdata/file.txt

exit

docker run --rm -it -v labvol:/appdata alpine:3.19 sh
-lc 'cat /appdata/file.txt'
```

2. **Using Bind Mounts (Host to Container)**

macOS/Linux:

```
mkdir bind && echo "host content" > bind/host.txt

docker run --rm -it -v $(pwd)/bind:/mnt alpine:3.19 sh
-lc 'cat /mnt/host.txt'
```

Windows:

```
mkdir bind | Out-Null

Set-Content -Path .\bind\host.txt -Value 'host content'

docker run --rm -it -v ${PWD}\bind:/mnt alpine:3.19 sh
-lc "cat /mnt/host.txt"
```

Observation / Output

- Pulled and inspected Alpine image.
- Created, modified, and committed container snapshots.
- Used Docker volumes and bind mounts for persistent storage.
- Successfully managed Docker image and container content.

Result

Explored and executed Docker commands for managing container images, container content, and persistent storage using volumes and mounts.

Conclusion

Docker provides powerful CLI commands for content and data management.
By understanding image, container, and volume management, developers can efficiently control application environments and data persistence.

Experiment 7: Develop a Simple Containerized Application using Docker

Objective

To create and run a simple **containerized web application** using Docker — serving an HTML page via NGINX.

Software / Tools Required

- Docker Engine (Desktop or CLI)
- Browser (for testing)
- Basic text editor (VS Code / Nano)

Theory

A containerized application packages the **application code**, **dependencies**, and **runtime environment** into a single unit.
This ensures that the app runs consistently across all environments (development, testing, production).

Here, Docker is used to deploy a **static event registration webpage** served by an NGINX web server inside a container.

Procedure

1. **Create project structure**

```
event-app/
├── Dockerfile
└── event_registration.html
```

2. **Create event_registration.html**
3. **Create Dockerfile**

```
# Use lightweight nginx image
FROM nginx:alpine

# Copy HTML into nginx default folder
COPY event_registration.html /usr/share/nginx/html/index.html

# Expose port 80
EXPOSE 80
```

```
# Start nginx (default CMD)
CMD ["nginx", "-g", "daemon off;”]
```

4. **Build the Docker image**

```
docker build -t event-app:1.0 .
```

5. **Run the container**

```
docker run -d -p 8000:80 --name event-container event-app:1.0
```

6. **Test the application**
    - Open a browser and go to:

      👉 **http://localhost:8000**

    - You should see your **Event Registration Page**.

Output

- Docker image built successfully.
- NGINX container served HTML page on port 8000.
- Verified web page through browser access.

Result

A simple HTML application was successfully containerized and deployed using Docker and NGINX.

Conclusion

Docker simplifies application deployment by encapsulating code, dependencies, and configurations. Even a basic static HTML file can be containerized and served consistently across environments — forming the foundation for deploying full-scale web applications.

Experiment 8: Integrate Kubernetes and Docker

Objective

To integrate Docker and Kubernetes for running, managing, and scaling containerized applications.

Software / Tools Required

- Docker Desktop / Docker Engine
- Kubernetes (via Docker Desktop, Minikube, or K3s)
- kubectl CLI tool
- Text editor (VS Code, Nano, etc.)

Theory

Docker and Kubernetes complement each other:

- **Docker** → builds and packages applications into containers.
- **Kubernetes (K8s)** → deploys, scales, and manages those containers automatically.

Integration means:
You build images using Docker, and Kubernetes uses those images to run containers inside Pods.
Docker acts as the **build system**, Kubernetes as the **orchestrator**.

Architecture Overview

```
[Source code + Dockerfile]
        ↓
docker build
        ↓
docker push (optional)
        ↓
kubectl apply -f k8s.yaml
```

Kubernetes Master Node manages worker Nodes:

- Master schedules Pods and manages desired state.
- Nodes actually run containers (via Docker/containerd runtime).

Procedure

Step 1 — Containerize the Application (from Experiment 7)

Dockerfile

```
FROM nginx:alpine
COPY event_registration.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Build the image

```
docker build -t event-app:1.0 .
```

Verify

```
docker run -d -p 8080:80 event-app:1.0
```

Access: http://localhost:8080

Then stop:

```
docker stop $(docker ps -q --filter ancestor=event-app:1.0)
```

Step 2 — Create Kubernetes Manifest (k8s.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: event-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: event-app
  template:
    metadata:
      labels:
        app: event-app
    spec:
      containers:
```

```
      - name: event-app
        image: event-app:1.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: event-app-service
spec:
  selector:
    app: event-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: NodePort
```

Step 3 — Deploy on Kubernetes

```
kubectl apply -f k8s.yaml
```

This creates:

- **Deployment** (manages replicas)
- **Service** (exposes app via NodePort)

Step 4 — Verify Deployment

```
kubectl get pods
kubectl get svc
```

Sample output:

```
NAME                    READY   STATUS    RESTARTS   AGE
event-app-deployment-7f5b5b7df6-tjw8k   1/1   Running   0
1m

NAME                TYPE        CLUSTER-IP      PORT(S)
AGE
event-app-service   NodePort    10.98.215.44    80:32244/TCP
1m
```

Step 5 — Access the Application

- **If using Minikube:**

  ```
  minikube service event-app-service
  ```

- **If using Docker Desktop:**

  Open → `http://localhost:<NodePort>` (e.g. http://localhost:32244)

✅ You'll see your `event_registration.html` page served from Kubernetes.

Cleanup

- **To delete all resources:**
  ```
  kubectl delete -f k8s.yaml
  ```

- **To temporarily stop app (scale down):**
  ```
  kubectl scale deployment event-app-deployment --
  replicas=0
  ```

Result

Successfully integrated Docker and Kubernetes — built a Docker image, deployed it as a Kubernetes Deployment, and accessed the containerized application via Service.

Conclusion

Docker packages the application, and Kubernetes deploys and manages it in a cluster. This integration ensures **portability, scalability, and automation** across environments.

Experiment 9: Automate Running the Containerized Application using Kubernetes

Objective

To automate the process of building, deploying, and running a Dockerized application on Kubernetes using shell scripts and automation tools.

Software / Tools Required

- Docker Desktop / Minikube with Kubernetes enabled
- kubectl CLI
- Bash / PowerShell
- (Optional) Skaffold or GitHub Actions for CI/CD automation

Theory

Automation removes manual steps like repeatedly building and deploying images.
With automation:

- Every code change triggers rebuild and redeploy.
- Ensures consistency between Docker image versions and Kubernetes Pods.

We achieve this using:

1. **Shell Script** → local build + deploy automation.
2. **Skaffold** → live rebuild/redeploy automation.
3. **GitHub Actions** → CI/CD automation on code push.

Procedure

Step 1 — Project Structure

```
event-app/
├── event_registration.html
├── Dockerfile
├── k8s.yaml
└── deploy.sh
```

Step 2 — Create Automation Script (`deploy.sh`)

```
#!/bin/bash

# Build Docker image
docker build -t event-app:1.0 .

# Load into Minikube (optional)
minikube image load event-app:1.0

# Deploy on Kubernetes
kubectl apply -f k8s.yaml

# Display running resources
kubectl get pods
kubectl get svc

# Open the app in browser (Minikube only)
minikube service event-app-service
```

Step 3 — Make Script Executable

```
chmod +x deploy.sh
```

Run it using:

```
./deploy.sh (on macOS/Linux)
or
bash deploy.sh (on Windows)
```

Step 4 — Output

You'll see:

```
📦 Building Docker image...
☸️  Deploying to Kubernetes...
✅ Pods Running: event-app-deployment-xxxx
🌐 Opening Service in browser...
```

Browser automatically opens showing your **event registration page**.

Result

Automated build and deployment pipeline created successfully.
Running `./deploy.sh` (or using Skaffold) rebuilds the Docker image and redeploys the updated container automatically in Kubernetes.

Conclusion

Automation simplifies the Kubernetes deployment workflow, ensuring continuous delivery of containerized applications.
By integrating Docker, Kubernetes, and automation scripts, the deployment process becomes faster, repeatable, and reliable.

Experiment 10: Install & Explore Selenium for Automated Testing

Objective

To install and explore Selenium WebDriver — an automation tool for testing web applications — on both macOS and Windows platforms using Python.

Software / Tools Required

- Google Chrome browser
- ChromeDriver
- Selenium WebDriver (JavaScript client library)
- Python
- macOS (Intel / Apple Silicon) or Windows 10/11 or Linux machine

Theory

Selenium is an open-source framework for automating browsers.
It allows developers and testers to simulate user interactions — clicking, typing, submitting forms, and verifying page behavior — for web-based functional testing.

Key Components:

1. **Selenium WebDriver** – Controls browsers through code.
2. **Selenium IDE** – A simple record-and-playback tool.
3. **Selenium Grid** – Executes tests on multiple machines and browsers in parallel.

Procedure

A. Install Prerequisites

Windows

1. **Install Python**
   Download and install from https://www.python.org/downloads/.
   During setup, select **"Add Python to PATH"**.

```
python --version
pip --version
```

2. **Install Google Chrome**
   Download from https://www.google.com/chrome/ .

3. **Install ChromeDriver**
   Option 1: Manually download ChromeDriver that matches your Chrome version from: https://googlechromelabs.github.io/chrome-for-testing/

   Option 2 (easier): Use pip package:
   ```
   pip install chromedriver-autoinstaller
   ```

🍎 macOS

1. **Install Homebrew (if not installed):**

   ```
   /bin/bash -c "$(curl -fsSL https://
   raw.githubusercontent.com/Homebrew/install/HEAD/
   install.sh)"
   ```

2. **Install Python (if not installed):**

   ```
   brew install python
   python3 --version
   pip3 --version
   ```

3. **Install Google Chrome and ChromeDriver:**

   ```
   brew install --cask google-chrome
   brew install chromedriver
   chromedriver --version
   ```

🐧 Linux (Ubuntu/Debian)

1. Install Python

```
sudo apt update
```

```
sudo apt install -y python3 python3-pip google-chrome-
stable
pip3 install selenium
```

2. If Chrome is not found:

```
wget https://dl.google.com/linux/direct/google-chrome-
stable_current_amd64.deb
sudo apt install ./google-chrome-stable_current_amd64.deb
```

3. Then install ChromeDriver:

```
sudo apt install -y chromium-chromedriver
```

B. Install Selenium Library

Run the following in **Command Prompt / Terminal**:

```
pip install selenium
```

Check the version:

```
python -m pip show selenium
```

C. Verify Installation with a Minimal Script

Create a file: **test_setup.py**

```
# test_setup.py
from selenium import webdriver

# Initialize Chrome WebDriver
driver = webdriver.Chrome()

# Open a webpage
driver.get("http://selenium.dev")

# Print the page title
print("Title:", driver.title)

# Close the browser
```

```
driver.quit()
```

D. Run the Script

```
python test_setup.py
```

Output:

1. A Chrome browser opens.
2. It navigates to `https://selenium.dev`
3. The console prints the page title:

```
Title: Selenium
```

Observation

Selenium successfully automated the Chrome browser — launching it, visiting a web page, and retrieving the title.

Result

Selenium WebDriver was successfully installed and tested with Python, demonstrating browser automation.

Conclusion

Selenium WebDriver provides a cross-platform automation interface for browser testing.
Installing and verifying Selenium on both macOS and Windows ensures readiness for developing automated test scripts in later experiments.

Experiment 11: Write a Simple JavaScript(or Python) Program and Test it using Selenium

Objective

To write a simple Selenium WebDriver test script that automates opening a webpage and validating the title.

Software / Tools Required

- Selenium WebDriver (JavaScript client)
- Google Chrome Browser
- ChromeDriver
- Python
- macOS / Windows

Theory

Selenium WebDriver allows programmatic control of a browser — navigating to URLs, filling forms, clicking elements, and verifying web content.
When integrated with JavaScript (Node.js) or python (pip), it enables developers to create **automated browser tests** that mimic real user actions.

Selenium interacts with browsers using a driver interface — in this case, **ChromeDriver** for the Chrome browser.

Procedure

A. Setup Environment

(Prerequisites should already be completed from **Experiment 10**.)

Ensure Chrome and ChromeDriver are properly installed and on PATH.

B. Write a Simple Test Script

Create a file named **`seleniumTest.py`** and add the following code:

```
// seleniumTest.py
```

```
from selenium import webdriver

# Initialize Chrome WebDriver
driver = webdriver.Chrome()

# Open a webpage
driver.get("http://selenium.dev")

# Print the page title
print("Title:", driver.title)

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

C. Execution Steps

🪟 Windows

```
pip install selenium
# Ensure chromedriver.exe is in PATH
python google_search_test.py
```

🍎 macOS

```
pip3 install selenium
brew install chromedriver
python3 google_search_test.py
```

🐧 Linux

```
sudo apt install chromium-chromedriver
pip3 install selenium
python3 google_search_test.py
```

Expected Behavior:

1. A Chrome browser opens.
2. It navigates to `https://selenium.dev`

3. The console prints the page title:

```
Title: Selenium
```

Output

```
Page title: Selenium
```
Browser closes automatically after the test.

Result:

A simple Python program was written and tested successfully using Selenium WebDriver, demonstrating browser automation and element interaction.

Conclusion

This experiment shows how Selenium WebDriver can be used with Python to automate real browser actions.
Such scripts can be expanded into test suites for validating web applications, integrated with CI/CD pipelines, and executed in headless mode for continuous testing.

Experiment 12. Develop Test Cases for the Containerized Application using Selenium

Objective

To develop and execute automated test cases using **Selenium WebDriver** for a containerized web application, ensuring the correctness of its functionality and deployment consistency.

Software / Tools Required

- **Docker** – for containerization
- **pip** – for running Selenium scripts
- **Selenium WebDriver** – for browser automation
- **Chrome & ChromeDriver** – for browser-based testing
- **Nginx** – to serve the HTML application inside a container

Description

The web application (`Event_registration.html`) is a simple event registration form hosted inside a Docker container using **Nginx**.
We use **Selenium WebDriver** (with Python) to automate the form submission and validate the success message.
This ensures the containerized application behaves correctly under automated testing.

Test Environment Setup

Create Dockerfile

```
FROM nginx:alpine

COPY Event_registration.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;”]
```

Build and Run the Container

```
docker build -t event-app:1.0 .

docker run -d -p 8000:80 --name event-container event-app:1.0
```

Install Selenium Dependencies

```
pip install selenium-webdriver chromedriver
```

Run Selenium Test

```
python3 registrationTest.py
```

Selenium Test Script (registrationTest.py)

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time

# Open browser
driver = webdriver.Chrome()

try:
    driver.get("http://localhost:8000")

    # Test 1: Check title
    print("Page Title:", driver.title)

    # Test 2: Fill form fields
    driver.find_element(By.ID, "fullname").send_keys("John
Doe")
    driver.find_element(By.ID,
"email").send_keys("john@example.com")
    driver.find_element(By.ID,
"phno").send_keys("9876543210")
    driver.find_element(By.CSS_SELECTOR,
"input[value='Male']").click()
    driver.find_element(By.ID, "event").send_keys("Workshop")
    driver.find_element(By.ID,
"address").send_keys("Hyderabad")

    # Submit the form
    driver.find_element(By.CSS_SELECTOR,
"input[type='submit']").click()
    time.sleep(1)

    # Handle alert
    alert = driver.switch_to.alert
    print("Alert Message:", alert.text)
    print("✅ Test Passed")
```

```
    alert.accept()

except Exception as e:
    print("❌ Error:", e)

finally:
    driver.quit()
```

Test Cases

Test No	Description	Expected Result
1	Open web page	Page loads successfully
2	Fill form fields	Data entered correctly
3	Submit form	Form submitted without error
4	Verify alert	Shows “Registration Successful”

Output

Docker container started successfully.

Selenium automated the form filling process.

Console output:

```
Alert message: Registration Successful!

✅ Test Passed
```

Result

The containerized web application was successfully tested using Selenium WebDriver.
All functional test cases passed, confirming that the application behaves correctly within the containerized environment.

Conclusion

Selenium provides an effective way to automate browser-based testing for applications deployed in Docker containers.
This approach ensures consistency, reliability, and repeatability of test execution in DevOps workflows.