Practical no 1

Q.1) create a simple project using any programming language and perform some operation on the project using git. Make your project as a git repository add your file staging area and commit changes with a descriptive message

Step 1: Create a simple Python project

```
mkdir simple-arithmetic-project

cd simple-arithmetic-project
```

Create a Python file:

```
# arithmetic.py

def add(x, y):

    return x + y

def subtract(x, y):

    return x - y

def multiply(x, y):

    return x * y

def divide(x, y):

    if y == 0:

        return "Error! Division by zero."

    return x / y

# Test the functions

if __name__ == "__main__":

    a = 10

    b = 5

    print(f"{a} + {b} = {add(a, b)}")

    print(f"{a} - {b} = {subtract(a, b)}")

    print(f"{a} * {b} = {multiply(a, b)}")

    print(f"{a} / {b} = {divide(a, b)}")
```

output:

```
    10   5 = 15     10 - 5 = 5    10 * 5 = 50      10 / 5 = 2.0
```

Step 2: Initialize Git Repository

Initialize a new Git repository:

```
git init
```

1) **Check the status of your repository:**
   ```
   git status
   ```

Step 3: Add Files to Git Staging Area

```
git add arithmetic.py
git status
```

Step 4: Commit Changes with a Descriptive Message

1) **Commit the changes**:

```
git commit -m "Initial commit: Added basic arithmetic operations (add, subtract, multiply, divide)"
```

2) **Check the commit log** to verify that the commit was successful:
   ```
   git log
   ```

Step 5: Push to a Remote Repository

1) Create a new repository on GitHub
2) Add the remote repository:
   ```
   git remote add origin https://github.com/yourusername/simple-arithmetic-project.git
   ```
3) **Push your changes to GitHub:**
   ```
   git push -u origin master
   ```

Q.2) create a simple Java project using Maven. adding dependencies, and Configuring the project's POM file and compile code using maven tool.

Step 1: Set Up the Project Directory

1) **Create a new directory for your Maven project:**
   ```
   mkdir simple-maven-java-project
   cd simple-maven-java-project
   mkdir -p src/main/java/com/example
   ```

Step 2: Initialize Maven Project

```
mvn archetype:generate -DgroupId=com.example -DartifactId=simple-maven-java-project -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Step 3: Understand and Configure the pom.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>simple-maven-java-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Example dependency (JUnit) -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

4)Add more dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>simple-maven-java-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Example dependency (JUnit) -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

5)Modify the Java Code

```
package com.example;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
package com.example;

import org.apache.log4j.Logger;
```

```
public class App {
    // Create a Logger instance
    final static Logger logger = Logger.getLogger(App.class);


    public static void main(String[] args) {
        // Log an informational message
        logger.info("Hello World from Log4j!");
    }
}
```

6) Compile and Build the Project with Maven

1. **Open your terminal/command prompt** and navigate to your project folder (if you're not already in it):

```
cd path/to/simple-maven-java-project
```

Step 7: Package the Project into a JAR File

To package your project into a .jar file:

```
mvn package
```

Step 8

: Run the Project

1) Add the following plugin configuration in the pom.xml file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.6.0</version>
            <executions>
                <execution>
                <goals>
                  <goal>java</goal>
                </goals>
                <configuration>
                  <mainClass>com.example.App</mainClass>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
```

1. **Run the project**

```
mvn exec:java
```

Practical no.2

Q.1) Docker and Containerization Task 1: Dockerfile Creation and Build ● Create a Dockerfile to containerize a simple HTML web page. ● The Dockerfile should use an nginx base image and copy the HTML page to the default directory served by nginx. ● Build the Docker image and tag it appropriately. Task 2: Running and Managing Containers ● Run the Docker container, mapping port 8080 on the host to the container's port 80. ● Confirm the web page is accessible through localhost:8080. ● Stop and remove the container after testing.

step 1: Dockerfile Creation and Build

```
mkdir simple-webpage

cd simple-webpage
```

Create an index.html

```
<!-- index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <meta name="viewport" content="width=device-width, initial-scale=1.0">

   <title>Welcome to My Simple Web Page</title>

</head>

<body>

   <h1>Hello, Dockerized Web Page!</h1>

   <p>This page is served using nginx in a Docker container.</p>

</body>

</html>
```

1.2: Create the Dockerfile

```
FROM nginx:latest

COPY index.html /usr/share/nginx/html/index.html

EXPOSE 80
```

1.3: Build the Docker Image

```
docker build -t simple-webpage-nginx .
```

step 2: Running and Managing Containers

```
docker run -d -p 8080:80 --name simple-webpage simple-webpage-nginx
```

1) **Confirm the Web Page is Accessible**
   http://localhost:8080

2) **web page displayed with the message:**

```
Hello, Dockerized Web Page!
This page is served using nginx in a Docker container.
```

3) **Stop and Remove the Container After Testing**

```
docker stop simple-webpage
```

```
docker rm simple-webpage
```

Q.2) create a simple project And push on remote server (like github) using git. and perform some operation. And displays a chronological history of commits.

Step 1: Create a Simple Java Project

Create a Simple Java File

App.java

```
public class App {

  public static void main(String[] args) {

    System.out.println("Hello, Welcome to my Simple Java Project!");

  }

}
```

Output:

```
Hello, Welcome to my Simple Java Project!
```

Step 2: Initialize Git and Commit

1) Initialize Git Repository
   `git init`
2) Check Git Status
   `git status`
3) Add Files to Staging Area
   `git add App.java`

4) Commit the Changes
   git commit -m "Initial commit: Added simple Java application"

Step 3: Set Up GitHub Repository

1) Create a New Repository on GitHub
2) Get the GitHub Repository UR

Step 4: Link Local Repository to GitHub and Push

git remote add origin https://github.com/yourusername/simple-java-project.git

Push the Local Repository to GitHub

git push -u origin master

Step 5: Modify the Project and Push Changes

1) App.java

```
public class App {

  public static void main(String[] args) {

    System.out.println("Hello, I have updated my Java project!");

  }

}
```

2) Check Git Status

git status

3) Stage the Changes
   git add App.java
4) Commit the Changes
   git commit -m "Update: Changed the message in App.java"
5) Push the Changes to GitHub
   git push origin master

Step 6: View the History of Commits

git log

practical no.3

Q.1) Applying CI/CD Principles to Web Development Using Jenkins, Git, and Local HTTP Server (e.g., Apache or Nginx).

Steps to Implement CI/CD Pipeline

1)Create a Project Folder

```
mkdir simple-web-app

cd simple-web-app

git init
```

2)Create an index.html file:

```
<!-- index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Simple Web App</title>

</head>

<body>

  <h1>Welcome to My Simple Web App!</h1>

  <p>This app is served using CI/CD pipeline with Jenkins, Git, and Apache/Nginx.</p>

</body>

</html>
```

3)Commit and Push to GitHub

```
git add .

git commit -m "Initial commit: Added simple HTML page"

git remote add origin https://github.com/yourusername/simple-web-app.git

git push -u origin master
```

Step 2: Set Up Jenkins for Continuous Integration

```
#!/bin/bash
```

```
echo "Building the project..."
```

Set Up Post-build Actions to Deploy to the Local HTTP Server

```
#!/bin/bash

echo "Deploying to Apache server..."

sudo cp -r * /var/www/html/

sudo systemctl restart apache2
```

Example script for Nginx:

```
#!/bin/bash

echo "Deploying to Nginx server..."

sudo cp -r * /usr/share/nginx/html/

sudo systemctl restart nginx
```

Q.2) Create a simple project, push it to a remote repository on GitHub, and create a new branch. Merge this branch into the main branch and display a chronological history of commits

Step 1: Create a Simple Project

1)Create a Simple File

```
print("Hello, this is my simple project!")
```

2)Initialize a Git Repository

```
git init
```

3)Stage and Commit the Initial File

```
git add app.py

git commit -m "Initial commit: Added simple Python app"
```

Step 2: Push the Project to GitHub

1)Create a Repository on GitHub

2)Add the Remote Repository

```
git remote add origin https://github.com/yourusername/simple-project.git
```

3)Push the Code to GitHub

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

Step 3: Create a New Branch

1)Create a New Branch

```
git checkout -b feature-branch
```

2)Make Some Changes in the New Branch

```
# app.py
print("Hello from the feature branch!")
```

3) Stage and Commit the Changes

```
git add app.py
git commit -m "Update: Changed message in feature branch"
```

Step 4: Push the New Branch to GitHub

1) Push the New Branch to GitHub

```
git push origin feature-branch
```

Step 5: Merge the Feature Branch into the Main Branch

1) Switch Back to the main Branch

```
git checkout main
```

2) Pull the Latest Changes from GitHub

```
git pull origin main
```

3) Merge the feature-branch into main

```
git merge feature-branch
```

4) Push the Merged Changes to GitHub

```
git push origin main
```

Step 6: Display a Chronological History of Commits

1) View the Commit History

```
git log --oneline --graph --decorate –all
```

practical no.4

Q.1) Configure Maven to compile the code, run tests, and generate artifacts like JAR files.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">


    <modelVersion>4.0.0</modelVersion>


    <!-- Project details -->

    <groupId>com.example</groupId>

    <artifactId>my-app</artifactId>

    <version>1.0-SNAPSHOT</version>

    <packaging>jar</packaging>


    <!-- Dependencies -->

    <dependencies>

        <!-- Example: JUnit dependency for testing -->

        <dependency>

            <groupId>junit</groupId>

            <artifactId>junit</artifactId>

            <version>4.13.2</version>

            <scope>test</scope>

        </dependency>

    </dependencies>


    <!-- Build configuration -->
```

```
<build>

  <plugins>

    <!-- Compiler plugin for Java version -->

    <plugin>

      <groupId>org.apache.maven.plugins</groupId>

      <artifactId>maven-compiler-plugin</artifactId>

      <version>3.8.1</version>

      <configuration>

        <source>1.8</source> <!-- Java version -->

        <target>1.8</target>

      </configuration>

    </plugin>


    <!-- JAR packaging plugin -->

    <plugin>

      <groupId>org.apache.maven.plugins</groupId>

      <artifactId>maven-jar-plugin</artifactId>

      <version>3.2.0</version>

      <configuration>

        <archive>

          <manifest>

            <mainClass>com.example.Main</mainClass> <!-- Main class for JAR -->

          </manifest>

        </archive>

      </configuration>

    </plugin>

  </plugins>
```

```
   </build>
</project>
```

Q.2) Create a simple project and use Git commands to check the status, view log history, see differences between the working directory and the last commit, make a commit and display a chronological history of commits.

Step 1: Create a Simple Project

1)Create a Directory:

```
  mkdir my-simple-project
 my-simple-project
```

2) Initialize Git:

```
git init
```

3)Create a Simple File:

```
echo "Hello, World!" > hello.txt
```

4)Add the File to Git:

```
git add hello.txt
```

4)Make an Initial Commit:

```
git commit -m "Initial commit: Add hello.txt"
```

Step 2: Use Git Commands

1)Check the Status:

```
git status
```

2)View Log History:

```
git log
git log –oneline
```

3)See Differences Between the Working Directory and the Last Commit:

```
echo "Adding a new line to check the diff" >> hello.txt
```

```
git diff
```

5)Make a New Commit

```
git add hello.txt
git commit -m "Update hello.txt with additional text"
```

6) Display a Chronological History of Commits:

```
git log --oneline --graph –all
```

Practical no.5

Q.1) Create a simple project, push it to a remote repository on BitBucket, and create a new branch. Merge this branch into the main branch and display a chronological history of commits.

Step 1: Set Up a Simple Project Locally

Create a Project Directory

1)Initialize a Git Repository:

```
git init
```

2)Create a File and Make an Initial Commit:

```
echo "Hello, Bitbucket!" > readme.txt
```

```
git add readme.txt
```

```
git commit -m "Initial commit: Add readme.txt"
```

Step 2: Push to a Remote Repository on Bitbucket

1)Create a Repository on Bitbucket:

2)Add Bitbucket as a Remote and Push:

```
git remote add origin https://bitbucket.org/your-username/my-bitbucket-project.git
```

```
git push -u origin main
```

Step 3: Create a New Branch

1)Create and Switch to a New Branch (e.g., feature-branch):

```
git checkout -b feature-branch
```

2)Make Changes in the New Branch:

```
echo "This is a new feature." >> readme.txt
```

```
git add readme.txt
```

```
git commit -m "Add new feature to readme.txt"
```

3)Push the New Branch to Bitbucket:

```
git push -u origin feature-branch
```

Step 4: Merge the Branch into the Main Branch

1)Switch Back to the Main Branch:

git checkout main

2)Merge the Feature Branch into Main:

git merge feature-branch

3)Push the Merged Main Branch to Bitbucket:

git push origin main

Step 5: Display a Chronological History of Commits

1)View Commit History:

git log --oneline --graph –all

Q.2) Create a simple project, push it to a remote repository on Github , and create a new branch. Merge this branch into the main branch and display a chronological history of commits.and Pull the changes on your local machine.

Step 1: Create a Simple Project

1)Create a Simple File

print("Hello, this is my simple project!")

2)Initialize a Git Repository

git init

3)Stage and Commit the Initial File

git add app.py

git commit -m "Initial commit: Added simple Python app"

Step 2: Push the Project to GitHub

1)Create a Repository on GitHub

2)Add the Remote Repository

git remote add origin https://github.com/yourusername/simple-project.git

3)Push the Code to GitHub

git push -u origin master

Step 3: Create a New Branch

1)Create a New Branch

git checkout -b feature-branch

2)Make Some Changes in the New Branch

```
# app.py
print("Hello from the feature branch!")
```

4) Stage and Commit the Changes

```
git add app.py
git commit -m "Update: Changed message in feature branch"
```

Step 4: Push the New Branch to GitHub

2) Push the New Branch to GitHub

```
git push origin feature-branch
```

Step 5: Merge the Feature Branch into the Main Branch

5) Switch Back to the main Branch

```
git checkout main
```

6) Pull the Latest Changes from GitHub

```
git pull origin main
```

7) Merge the feature-branch into main

```
git merge feature-branch
```

8) Push the Merged Changes to GitHub

```
git push origin main
```

Step 6: Display a Chronological History of Commits

1)View the Commit History

```
git log --oneline --graph --decorate –all
```

Step 7: Pull the Changes on Your Local Machine

1. **Pull the Latest Changes from GitHub**:
   git pull origin main

practical no.6

Que 1) =practical 5(1)
Que 2) =practical 5(2)

practical no.7

Q.1) =practical(5.1)

Q.2) Create CI using Webhook and deploy a project using Jenkins Execute shell.

Step 1: Set Up Your Project on GitHub

1)Create a simple project locally and push it to a GitHub repository:

```
mkdir my-ci-project
cd my-ci-project
echo "Hello, Jenkins CI!" > hello.txt
git init
git add hello.txt
git commit -m "Initial commit for CI project"
git remote add origin https://github.com/your-username/my-ci-project.git
git push -u origin main
```

Step 2: Set Up Jenkins

1. Install Jenkins on your server if it's not already installed.

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add
-sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
sudo apt update
sudo apt install jenkins
sudo systemctl start jenkins
```

Step 3: Create a Jenkins Job

1. Create a New Job in Jenkins:
2. Configure Source Code Management:
3. Build Triggers:
4. Add a Build Step:

```
echo "Starting deployment"
# Sample commands; replace these with your actual build and deploy commands
echo "Running tests"
echo "Deploying application"
```

5) Save the Job Configuration**.**

Step 4: Set Up a GitHub Webhook

1. Go to Your GitHub Repository:
2. Configure the Webhook:

2) Add the Webhook. This will allow GitHub to notify Jenkins of changes, triggering the CI process.

Step 5: Test the CI Pipeline

1)Make a Change in the GitHub Repository:

```
echo "CI pipeline test change" >> hello.txt

git add hello.txt

git commit -m "Testing CI with Jenkins"

git push origin main
```

practical no.8

Q.1) Create a new file on a separate branch, make some changes to this file, and then merge these changes into the main branch using bitBucket interface.

Step 1: Create a New Branch Locally and Add a New File

1)Clone the Bitbucket Repository (if you haven't already):

```
git clone https://bitbucket.org/your-username/your-repository.git
cd your-repository
```

2)Create and Switch to a New Branch:

```
git checkout -b new-feature-branch
```

3)Create a New File and Make Changes:

```
echo "This is a new feature file." > newfile.txt
git add newfile.txt
git commit -m "Add newfile.txt to the new feature branch"
```

4)Push the New Branch to Bitbucket:

```
git push -u origin new-feature-branch
```

Step 2: Create a Pull Request in the Bitbucket Interface

1)Go to the Bitbucket Repository:

2)Create a Pull Request:

3)Review the Pull Request:

4)Merge the Pull Request:

Step 3: Delete the Branch (Optional)

1)Delete the Branch in Bitbucket:

Step 4: Update Your Local Repository

1)Switch to the Main Branch Locally and Pull the Latest Changes:

```
git checkout main
git pull origin main
```

Q.2) Outline the process of setting up a CI/CD pipeline for a web application using Jenkins, Git, and a local HTTP server. Include the configuration of Jenkins, the webhook setup, and the execution of build and deployment steps.

Step 1: Create and Push Your Web Application to Git Repository

1)Create the Web Application Project:

2)Initialize a Git Repository:

```
git init
echo "Hello, CI/CD with Jenkins!" > index.html
git add index.html
git commit -m "Initial commit for web application"
```

3)Push to a Remote Repository (e.g., GitHub or Bitbucket):

```
git remote add origin https://github.com/your-username/your-webapp-repo.git
git push -u origin main
```

Step 2: Set Up Jenkins

1)Create a New Jenkins Job:

2)Configure Git Repository in Jenkins:

1)Configure Build Triggers:

Step 3: Set Up Webhook in Git Repository

1)Go to Your Git Repository (e.g., GitHub or Bitbucket):

1)Add a Webhook:

Step 4: Configure Build and Deployment Steps in Jenkins

1)Add a Build Step:

Here is an example script:

```
cd /var/www/html/your-webapp-folder
rm -rf *
git clone https://github.com/your-username/your-webapp-repo.git .
```

```
# Optional: Run build commands, e.g., npm install, if it's a Node.js app

# npm install

# npm run build


echo "Deployment completed successfully!"
```

1. Save and Run the Jenkins Job:

Step 5: Test the Pipeline

1. Trigger the Pipeline:
2. Monitor the Job in Jenkins:

Step 6: Verify Web Application on Local HTTP Server

1. Check the Local HTTP Server

Practical no.9

Q.1) Implement Bitbucket Operations Using Git ● Task 1: Create a new repository on Bitbucket and clone it locally. ● Task 2: Create a file example.txt, add and commit it, and create a branch feature. ● Task 3: Push the feature branch to Bitbucket and create a pull request. Review and merge the pull request.

Task 1: Create a New Repository on Bitbucket and Clone it Locally

Create a New Repository on Bitbucket:

1)Clone the Repository Locally:

```
git clone https://bitbucket.org/your-username/my-bitbucket-repo.git
```

Change to the repository directory:

```
cd my-bitbucket-repo
```

Task 2: Create a File example.txt, Add and Commit It, and Create a Branch

1)Create a File:

```
echo "This is an example file." > example.txt
```

2)Add and Commit the File:

```
git add example.txt
git commit -m "Add example.txt"
```

3)Create a New Branch feature:

```
git checkout -b feature
```

Task 3: Push the feature Branch to Bitbucket and Create a Pull Request

1)Push the feature Branch to Bitbucket:

```
git push -u origin feature
```

2)Create a Pull Request:

Go to Bitbucket and navigate to your repository.

Switch to the **Branches** tab, find the feature branch, and click on **Create pull request**.

Add a title and description for the pull request.

Review the changes (Bitbucket will show the diff), and click **Create pull request**.

3)Review and Merge the Pull Request:

Q.2) Explain how to set up and manage dependencies in a Maven project. Describe the structure of a pom.xml file, adding dependencies, and how Maven handles build automation for tasks like compiling code, running tests, and generating JAR files.

Structure of pom.xml File

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>      <!-- Organization or project namespace -->
  <artifactId>my-maven-project</artifactId> <!-- Unique project identifier -->
  <version>1.0-SNAPSHOT</version>     <!-- Project version -->

  <dependencies>                <!-- Project dependencies -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>       <!-- Used only for tests -->
    </dependency>
  </dependencies>

  <build>                     <!-- Build configuration -->
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
```

```
            <version>3.8.1</version>
            <configuration>
               <source>1.8</source> <!-- Java source version -->
               <target>1.8</target> <!-- Java target version -->
            </configuration>
         </plugin>
      </plugins>
   </build>
</project>
```

<dependencies> Section

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.3.8</version>
   </dependency>
</dependencies>
```

2)Adding Dependencies

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
   <version>5.3.8</version>
</dependency>
```

3)Managing Plugins for Build Automation

```
<build>
   <plugins>
      <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

How Maven Handles Dependencies and Builds

Maven simplifies dependency management by downloading required libraries automatically, based on the information in the pom.xml. It also maintains these dependencies in a local repository, avoiding redundancy and ensuring consistency across projects.

Practical no.10

Q.1) Install Docker on your system and create a simple "Hello, World!" application using HTML. Create a Dockerfile to containerize the application, using an official web server image as the base. Build the Docker image, tag it, and run a container, making the application accessible on a local port (e.g., http://localhost:8080).

Step 1: Install Docker

1)For Windows/macOS:

- Download and install Docker Desktop from the official website: Docker Desktop.
- Follow the installation instructions for your OS.
- Once installed, start Docker Desktop.

2) **For Linux**:

```
sudo apt update

sudo apt install apt-transport-https ca-certificates curl software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"

sudo apt update

sudo apt install docker-ce
```

Verify Docker installation:

```
sudo docker --version

sudo systemctl enable --now docker
```

Step 2: Create a Simple "Hello, World!" HTML Application

1)Create a Project Directory

```
mkdir hello-docker

cd hello-docker
```

2)Create the HTML File

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Hello, World!</title>
</head>
<body>
  <h1>Hello, World!</h1>
  <p>Welcome to my Dockerized application!</p>
</body>
</html>
```

Step 3: Create the Dockerfile

1)Create a Dockerfile:

```
# Use the official Nginx image as the base
FROM nginx:latest
# Copy the HTML file into the container's Nginx web server directory
COPY index.html /usr/share/nginx/html/index.html
# Expose port 80 to be accessible from outside the container
EXPOSE 80
```

Step 4: Build the Docker Image

Copy code

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

Tag the Image

Copy code

```
docker tag hello-world-app hello-world-app:v1
```

Step 5: Run the Docker Container

Copy code

```
docker run -d -p 8080:80 hello-world-app
```

Verify the Container is Running: To check if the container is running, use the following command:

Copy code

```
docker ps
```

Step 6: Access the Application

1)Open a Web Browser:

Step 7: Stop and Clean Up

1)Stop the Running Container

```
docker ps      # Find the container ID or name
docker stop <container_id>
```

2)Remove the Container

```
docker rm <container_id>
```

3)Remove the Image

```
docker rmi hello-world-app
```

Q.2) Git and GitHub Repository Management. Task 1: Repository Setup and Initial Commit ● Set up a local Git repository and create a file named project.md with a brief description of a hypothetical project. ● Initialize the repository, add project.md, commit the changes, and push to a GitHub repository. Task 2: Branching and Merging ● Create a new branch called feature-branch and make additional changes to project.md. ● Commit the changes in the feature-branch, switch back to main, and merge feature-branch into main. ● Push the updated main branch to GitHub, ensuring the merge is reflected.

Task 1: Repository Setup and Initial Commit

1. Set Up a Local Git Repository

1)Create a Directory for the Project: Start by creating a directory for your project and navigating into it.

```
mkdir my-project
cd my-project
```

2)Initialize the Git Repository: Initialize a new Git repository in this directory.

git init

3)Create project.md: Create a new file named project.md and add a brief description of your hypothetical project.

echo "# My Hypothetical Project" > project.md

echo "This is a simple project for learning Git and GitHub repository management." >> project.md

4)Add project.md to the Git Repository: Stage the project.md file to be added to the repository.

git add project.md

5)Commit the Changes: Commit the staged file to the local repository with a meaningful commit message.

git commit -m "Initial commit with project.md"

1. **Create a GitHub Repository**:
2. **Push the Local Repository to GitHub**: Connect your local repository to the GitHub repository and push the initial commit.

git remote add origin https://github.com/your-username/my-project.git

git branch -M main # Rename the default branch to "main" if needed

git push -u origin main

Replace your-username with your actual GitHub username.

Task 2: Branching and Merging

1. Create a New Branch (feature-branch)

1)Create and Switch to a New Branch: Create a new branch called feature-branch and switch to it.

git checkout -b feature-branch

2)Make Changes to project.md: Edit project.md to add more details about the project.

echo "This feature branch adds more information to the project." >> project.md

3)Stage and Commit the Changes: Stage the changes and commit them with a message.

git add project.md

git commit -m "Update project.md with additional details in feature-branch"

2. Switch Back to the main Branch and Merge

1)Switch to the main Branch: Switch back to the main branch.

```
git checkout main
```

2)Merge feature-branch into main: Merge the feature-branch into the main branch.

```
git merge feature-branch
```

3.)Push the Updated main Branch to GitHub

1)Push the Changes to GitHub: After merging, push the updated main branch to GitHub.

```
git push origin main
```

1)Verify on GitHub:

Practical no.11

Q.1) Containerize a basic application and deploy it using Docker

Step 1: Create a Basic Application

1)Set up the project directory:

```
mkdir my-node-app

cd my-node-app
```

1)Create a package.json file to define the app's dependencies:

```
npm init -y
```

1)Install Express (a lightweight Node.js framework):

```
npm install express
```

1)Create a server.js file that will run a basic HTTP server:

```
// server.js

const express = require('express');

const app = express();

const port = 3000;

app.get('/', (req, res) => {

   res.send('Hello, World!');

});

app.listen(port, () => {

   console.log(`Server is running at http://localhost:${port}`);

});
```

1)Run the application locally (optional):

```
node server.js
```

Visit http://localhost:3000 to ensure the app is working.

Step 2: Create a Dockerfile

1)Create a Dockerfile in the root of your project directory:# Use the official Node.js image from Docker Hub as the base image

```
# Set the working directory inside the container

WORKDIR /usr/src/app

# Copy the package.json and package-lock.json files into the container

COPY package*.json ./

# Install the application dependencies

RUN npm install

# Copy the rest of the application files into the container. .

# Expose the port the app will run on

EXPOSE 3000

# Define the command to run the application

CMD ["node", "server.js"]
```

Step 3: Build the Docker Image

1)Build the Docker image by running the following command in the project directory:

```
docker build -t my-node-app .
```

2)Verify the image was created:

```
docker images
```

Step 4: Run the Application in a Docker Container

1)Run the container using the newly built image:

```
docker run -p 3000:3000 my-node-app
```

2**Verify the application is running**:.

Step 5: (Optional) Push the Docker Image to Docker Hub

1)**Log in to Docker Hub**

2)Tag the image

3)Push the image

Q.2)= practical(10.2)

Practical no.12

Q.1) Applying CI Principles to Web Development Using Jenkins, Git

Steps to Apply CI Principles to Web Development Using Jenkins and Git

1. Prerequisites

- Jenkins installed and configured on your system or on a CI server.
- Git repository for your web application (GitHub, GitLab, Bitbucket, etc.).
- A web development project with source code, such as a Node.js app, React app, or any other frontend application.
- A test suite for the web application (unit tests, integration tests, etc.).

Step 2: Set Up Jenkins

1)Install Jenkins:

2)Start Jenkins:

```
sudo systemctl start jenkins
```

1. **Access Jenkins**:
2. **Install Required Plugins**:

Jenkins requires plugins to integrate with Git and other tools. Install these plugins:

- **Git Plugin**:
- **NodeJS Plugin**).
- **Maven Plugin**
- **Pipeline Plugin**.

To install plugins, go to **Manage Jenkins** > **Manage Plugins** and search for the required plugins.

Step 3: Set Up a Git Repository

1. **Create a Git Repository**:

```
git clone https://github.com/your-username/your-web-project.git
```

2. **Push Code to Git**:

Develop your web application, make changes, and commit them to the Git repository:

```
git add .

git commit -m "Initial commit with web app code"

git push origin main
```

Step 4: Create a Jenkins Job or Pipeline

1. **Create a New Jenkins Job**:
   - Go to **Jenkins Dashboard** and click **New Item**.
   - Select **Freestyle project** (or **Pipeline** if you want to use Jenkins pipelines) and provide a name for the job
2. **Configure the Job**:
3. **Add Build Steps**:
   - Depending on the type of web project (Node.js, React, Java, etc.), you can add build steps. Common examples:

For Node.js/React:

Install Dependencies: Run npm install or yarn install.

Run Tests: Execute npm test or yarn test to run the test suite.

For Maven (Java Projects):

Add the **Maven build step** to compile, test, and package the application using Maven commands (e.g., mvn clean install).

4. **Example Build Commands**:

For **Node.js/React**:

```
npm install

npm test

npm run build  # If you want to generate a build folder for deployment
```

For **Java (Maven)**:

```
mvn clean install

mvn test
```

5. **Add Post-build Actions**:

You can add steps like sending email notifications, archiving test results, or deploying the build to a server or cloud platform.

For **Post-Build Actions**, you might use tools like **Deploy to AWS**, **FTP Deployment**, or deploy it to your web server using **Shell Scripts**.

Step 5: Set Up Webhook for Automatic Builds

Set Up a Webhook in GitHub/GitLab/Bitbucket:

Go to your Git repository (e.g., GitHub) and configure a **Webhook**.

The webhook should be configured to trigger Jenkins builds whenever changes are pushed to the repository. Provide the following URL to the Git webhook configuration:

http://<jenkins-server>/github-webhook/

This will allow Jenkins to automatically start a build when there is a change in the repository.

1)Configure Jenkins to Respond to the Webhook:

In Jenkins, you need to configure the job to listen for incoming webhooks.

Under the **Build Triggers** section of the Jenkins job, check the option **GitHub hook trigger for GITScm polling** (or **Build when a change is pushed to GitHub**).

Step 6: Run and Monitor the Builds

1)Triggering Builds:

After setting up the webhook, any push to the repository will automatically trigger a build in Jenkins.

You can also manually trigger a build by clicking **Build Now** on the Jenkins job page.

1)Monitor the Build Process:

2)**Test Results**:

Step 7: Automate Deployment (Optional)

Deploy to a Server:

FTP/SFTP for traditional web servers.

AWS Elastic Beanstalk or **Heroku** for cloud-based deployments.

Docker containers for deploying via Docker on a server or cloud.

1. **Add Deployment Steps in Jenkins**:

```
scp -r ./build/* user@server:/var/www/html/
```

Step 8: Continuous Monitoring and Feedback

Q.2)=practical (11.1)

Practical no.13

Q.1) Bitbucket Repository and Branch Management

Task 1: Repository Setup and Branching

● Create a repository on Bitbucket and clone it locally.

● Create a branch development, add a new file, commit the changes, and push it to the development branch on Bitbucket.

Task 2: Pull Request and Code Review

● In Bitbucket, create a pull request to merge development into main.

● Assign a reviewer (or self-review) and comment on any changes before merging.

Task 1: Repository Setup and Branching

1. Create a Repository on Bitbucket

1)Go to Bitbucket and log in to your account.

2)Click on Create Repository.

3)Fill in the repository details:

Repository Name: Choose a name for your repository (e.g., my-web-project).

Access Level: Choose whether the repository should be public or private.

Include a README: Optionally, you can initialize the repository with a README file (but we'll initialize it without a README in this task).

1)Click Create repository to create your new Bitbucket repository.

2). Clone the Repository Locally

```
git clone https://bitbucket.org/your-username/my-web-project.git
```

4.Navigate into the cloned repository**:**

```
cd my-web-project
```

3. Create a Branch Named development

1. Create a new branch named development and switch to it:

   ```
   git checkout -b development
   ```

2. Verify the branch creation:

3. git branch

. 4. Add a New File

1. **Create a new file, for example, new_feature.txt, and add some content to it:**

```
echo "This is a new feature on the development branch" > new_feature.txt
```

2. **Stage the new file for commit:**

```
git add new_feature.txt
```

3. **Commit the changes:**

```
git commit -m "Add new_feature.txt for the development branch"
```

4. Push the Changes to Bitbucket

```
git push -u origin development
```

Task 2: Pull Request and Code Review

1. Create a Pull Request to Merge development into main

1. Go to your Bitbucket repository in the browser.
2. Click on the Branches tab to view all branches.
3. You should see the development branch listed. Click on Create pull request next to the development branch.
4. In the pull request form:
   - Source branch: development (This is the branch you're merging from).
   - Destination branch: main (This is the branch you're merging into).
   - Add a title and a description for the pull request, such as:
     - Title: "Merge development into main"
     - Description: "Merging the new feature changes from the development branch into the main branch."
5. Once everything is filled out, click Create pull request.

2. Assign a Reviewer and Comment on Changes

1. In the pull request page, you can assign a reviewer. If you are working solo, you can self-assign as the reviewer. To do this:

   - In the Reviewers section, select your username to assign yourself as the reviewer.
   - If working in a team, you can select a teammate to review your code.
2. Comment on the Changes (Optional):
   - Bitbucket allows you to leave comments directly on the code changes in the pull request. You can review the code, suggest improvements, or ask questions.
   - If you want to comment on a specific line of code, simply click on the line number in the pull request diff view and add your comment.
3. Approve the Pull Request:
   - After reviewing the code and ensuring everything is fine, click Approve (if you're the reviewer) to indicate that the code is ready for merging.
   - If there are any necessary changes, you can leave comments and ask the developer to make adjustments before approving.

3. Merge the Pull Request

1. Once the pull request is reviewed and approved, you can merge the development branch into the main branch.
2. Click Merge on the pull request page.
3. Choose the merge strategy if prompted (default is usually fine). You can use the Merge commit (creates a merge commit) or Squash merge (combines all commits into one commit).
4. Click Merge to finalize the merge.

Q.2) Automated Deployment with Jenkins and GitHub Task 1: Configuring Jenkins with GitHub ● Install and set up Jenkins. ● Integrate Jenkins with a GitHub repository, ensuring Jenkins triggers a build on every push. Task 2: Creating a CI/CD Pipeline ● Create a Jenkins pipeline that clones your GitHub repository, builds a simple web application, and archives the build artifacts. ● Configure a post-build action to notify your GitHub repository of the build status. Task 3: Adding Deployment Step ● Extend the pipeline to deploy the built application to a local web server (use shell commands for deployment). ● Set up a webhook in GitHub to trigger the Jenkins job automatically upon a code push.

Task 1: Configuring Jenkins with GitHub

1. Install and Set Up Jenkins

```
sudo apt update

sudo apt install openjdk-11-jdk

sudo apt install Jenkins
```

2 **Start Jenkins**:

```
sudo systemctl start jenkins
```

3 **Unlock Jenkins**:

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

4 **Install Plugins**:

5 **Set Up Credentials**:

Go to **Manage Jenkins** > **Manage Credentials** > **(Global)** > **Add Credentials**.

Choose **Username with password** and enter your GitHub username and personal access token (instead of your password, use a GitHub token if 2FA is enabled)

Task 2: Creating a CI/CD Pipeline

1. Integrate Jenkins with GitHub Repository

1)Create a New Jenkins Job:

2)Configure GitHub Integration:

3) Pipeline Script:

```
pipeline {
  agent any


  environment {
    // Define environment variables if needed
  }


  stages {
    stage('Clone Repository') {
      steps {
            git url: 'https://github.com/your-username/your-web-app.git', branch: 'main'
        }
    }

    stage('Install Dependencies') {
        steps {
            sh 'npm install'  // For a Node.js app
        }
    }

    stage('Build Application') {
        steps {
            sh 'npm run build'  // Assuming there is a build command in package.json
        }
    }

    stage('Archive Build Artifacts') {
        steps {
            archiveArtifacts artifacts: 'build/**', allowEmptyArchive: true
        }
    }
}

post {
    success {
        // You can add additional actions, like notifying the GitHub repository of success.
        echo 'Build was successful!'
        }
        failure {
            echo 'Build failed.'
        }
    }
}
```

4)Save and Run the Job:

Task 3: Adding Deployment Step

1. Extend the Pipeline to Deploy the Application

1. **Add Deployment Stage**:

```
stage('Deploy to Server') {
    steps {
        sh '''
        scp -r ./build/* user@your-server:/path/to/your/web-server-directory/
        '''
    }
}
```

2)Full Jenkins Pipeline with Deployment:

```
  pipeline {
    agent any

    environment {
        // Define environment variables if needed
    }

    stages {
        stage('Clone Repository') {
```

```
    steps {
        git url: 'https://github.com/your-username/your-web-app.git', branch: 'main'
    }
}

stage('Install Dependencies') {
    steps {
        sh 'npm install'
    }
}

stage('Build Application') {
    steps {
        sh 'npm run build'
    }
}

stage('Archive Build Artifacts') {
    steps {
        archiveArtifacts artifacts: 'build/**', allowEmptyArchive: true
    }
}

stage('Deploy to Server') {
    steps {
        sh '''
        scp -r ./build/* user@your-server:/path/to/your/web-server-directory/
```

```
            '''
          }
        }
      }

      post {
        success {
          echo 'Build and deployment were successful!'
        }
        failure {
          echo 'Build or deployment failed.'
        }
      }
    }
```

2. Set Up GitHub Webhook to Trigger Jenkins Job Automatically

1)http://your-jenkins-server/github-webhook/

2)Configure Jenkins to Handle the Webhook

Practical no.14

Que 1=practical(13.1)

Que 2=practical(3.1)

Practical no.15

Q.1 =practical(13.1)

Q.2=practical(1.1)

Practical no.16

Q.1 =practical(1.2)

Q.2 =practical(3.1)

Practical no.17

Q.1 =practical(13.1)

Q.2 =practical(2.2)

Practical no.18

Q.1 =practical(12.1)

Q.2 =practical(13.1)

Practical no.19

Q.1 =practical(13.1)

Q.2 =practical(3.2)

Practical no.20

Q.1 =practical(12.1)

Q.2 =practical(7.1)

Practical no.21

Q.1 =practical(8.1)

Q.2 =practical(7.2)

Practical no.22

Q.1 =practical(1.2)

Q.2 =practical(10.1)

Practical no.23

Q.1 =practical(12.1)

Q.2 =practical(3.2)

Practical no.24

Q.1 =practical(7.2)

Q.2 =practical(16.2)

Practical no.25

Q.1 =practical(3.2)

Q.2 =practical(7.2)