Ex. No. 1	Git & GitHub
12/07/2024	

AIM

Algorithm for Git and GitHub Operations

Algorithm

1. **Create a Repository**:
    - Initialize a new Git repository named "Master".
    - Verify repository initialization.
2. **Add and Commit Changes**:
    - Create a new text file named `README.md`.
    - Add content to the `README.md` file.
    - Add the file to the staging area.
    - Commit the changes with a message.
3. **Exploring History**:
    - Modify or add content to the text file.
    - Add and commit the changes.
4. **Branching and Merging**:
    - Create a new branch named `Updated_ReadMe`.
    - Commit changes in the new branch.
    - Merge changes from `Updated_ReadMe` branch into `Master` branch.
5. **Collaborating with Remote Repositories**:
    - Link the local repository to a remote repository.
6. **Push Changes**:
    - Push local commits to the remote repository.
7. **Cloning a Repository**:
    - Clone the repository from GitHub.
8. **Making Changes and Creating a Branch**:
    - Navigate into the cloned repository.
    - Make necessary changes.
    - Check the status of the repository.
9. **Pushing Changes to GitHub**:

- Add the changes to the staging area.
- Commit the changes.
- Push the local branch to the remote repository.

10. **Collaborating through Pull Requests**:

- Create a pull request on GitHub.
- Choose base branch and compare with the new branch.
- Review and merge the pull request.
- Add title, description, and assign reviewers.
- Reviewers approve and merge the pull request into the base branch.

11. **Syncing Changes**:

- Update local repository after the pull request is merged.

Input

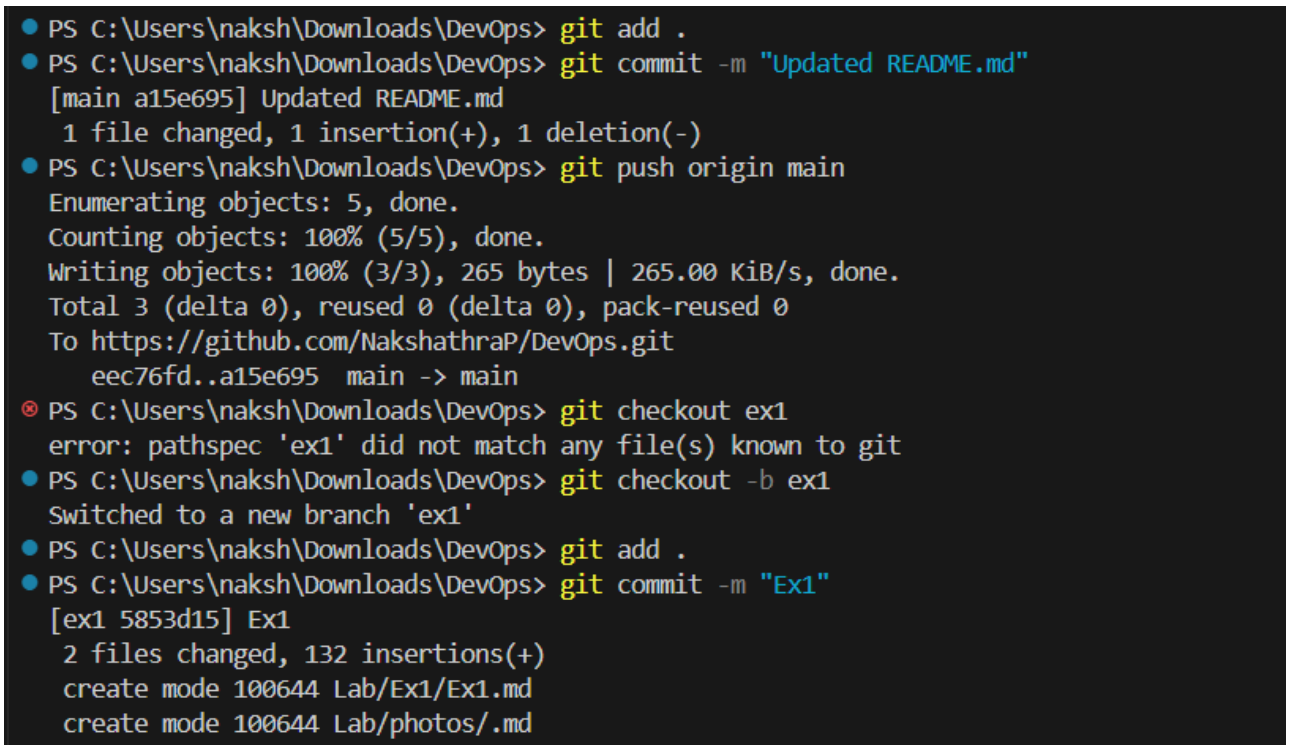
```
# Task 1
## a. Create a Repository
git init

## b. Add and Commit Changes
touch README.md
echo "# This is the readme file for devops" > README.md
git add README.md
git commit -m "Added README.md"

## c. Exploring History
echo "# This is the README file for DevOps" > README.md
git add README.md
git commit -m "Corrected content of README.md"

## d. Branching and Merging
git checkout -b Updated_ReadMe
echo "# This is the updated README file for DevOps" > README.md
git add README.md
git commit -m "Updated README.md"
git checkout main
git merge Updated_ReadMe

## e. Collaborating with Remote Repositories
git remote add origin https://github.com/NakshathraP/DevOps.git

## f. Push Changes
git branch -M main
git push -u origin main

# Task 2
## a. Cloning a Repository
git clone https://github.com/NakshathraP/DevOps.git
```

```
## b. Making Changes and Creating a Branch
cd WebTechLab
touch README.md
echo "# This is the README file for WebTechLab" > README.md
git status

## c. Pushing Changes to GitHub
git add .
git commit -m "Added README.md"
git push -u origin main

## d. Collaborating through Pull Requests
# Performed on GitHub:
# 1. Create a pull request.
# 2. Choose base branch and compare with new branch.
# 3. Review and merge the pull request.
# 4. Add title and description.
# 5. Assign reviewers.
# 6. Reviewers approve and merge the pull request.

## e. Syncing Changes
git pull origin main
```

Output

1. **Repository Creation**:
    - A new Git repository named "Master" is created.
    - Confirmation message displayed.
2. **Add and Commit Changes**:
    - `README.md` file is created.
    - File contents are added.
    - File is added to the staging area and committed.
3. **Exploring History**:
    - `README.md` file is modified and changes committed.
4. **Branching and Merging**:
    - New branch `Updated_ReadMe` is created and changes committed.
    - Changes are merged into the `main` branch.
5. **Collaborating with Remote Repositories**:
    - Local repository is linked to the remote repository.
6. **Push Changes**:
    - Local commits are pushed to the remote repository.

7. **Cloning a Repository**:
    - Repository is cloned from GitHub.
8. **Making Changes and Creating a Branch**:
    - Changes are made and the status of the repository is checked.
9. **Pushing Changes to GitHub**:
    - Changes are pushed to the remote repository.
10. **Collaborating through Pull Requests**:
    - Pull request is created, reviewed, and merged on GitHub.
11. **Syncing Changes**:
    - Local repository is updated with the latest changes from the remote repository.



Ex. No. 2	Git & GitLab
07/08/2024	

AIM

Algorithm for Implementing GitLab Operations using Git

Algorithm

1. **Creating a Repository**:
    - Log into your GitLab account.
    - Click on the `New project` button.
    - Select `Create blank project`.
    - Fill in the project details and set the visibility level.
    - Click on the `Create project` button.
2. **Cloning a Repository**:
    - Copy the repository URL from GitLab.
    - Open the terminal.
    - Clone the repository using the URL.
3. **Making Changes and Creating a Branch**:
    - Navigate into the cloned repository.
    - Create a new text file named `test.txt`.
    - Add content to the `test.txt` file.
    - Check the repository status.
    - Stage the changes for commit.
    - Commit the changes with a message.
    - Create a new branch named `feature`.
    - Switch to the `feature` branch.
4. **Pushing Changes to GitLab**:
    - Add the repository URL to a variable.
    - Push the `feature` branch to GitLab.
    - Confirm the new branch is available in the GitLab repository.
5. **Collaborating through Merge Requests**:
    - Go to the `Merge Requests` tab in the GitLab repository.
    - Click on the `New merge request` button.
    - Select the `source branch` as `feature` and the `target branch` as `main`.
    - Click on the `Compare branches and continue` button.
    - Fill in the merge request details and create the merge request.
    - Review and merge the request.
6. **Syncing Changes**:

- After merging, update the local repository with the latest changes from the remote repository.

Input

```
# Task 1: Creating a Repository
# 1. Log into your GitLab account.
# 2. Click on the `New project` button.
# 3. Select `Create blank project`.
# 4. Fill in the project details:
#    - Project name: `DevOpsLab`
#    - Project slug: `devopslab` (auto-filled)
#    - Visibility level: Choose between Private, Internal, or Public.
# 5. Click on the `Create project` button.

# Task 2: Cloning a Repository
# 1. Copy the repository URL.
# 2. Open terminal.
git clone https://gitlab.com/MukeshTheGreat/devopslab.git

# Task 3: Making Changes and Creating a Branch
cd devopslab
touch test.txt
echo "This is a test file." > test.txt
git status
git add test.txt
git commit -m "Added test.txt"
git branch feature
git checkout feature

# Task 4: Pushing Changes to GitLab
REPO_URL=https://gitlab.com/MukeshTheGreat/devopslab.git
git push -u origin feature

# Task 5: Collaborating through Merge Requests
# 1. Go to the GitLab repository.
# 2. Click on the `Merge Requests` tab.
# 3. Click on the `New merge request` button.
# 4. Select `source branch` as `feature` and `target branch` as `main`.
# 5. Click on the `Compare branches and continue` button.
# 6. Fill in the merge request details and click `Create merge request`.
# 7. Review and click `Merge` to merge the request.

# Task 6: Syncing Changes
git checkout main
git pull origin main
```

Output

1. **Creating a Repository**:

- A new GitLab repository named `DevOpsLab` is created with the specified details.  1-1  1-2  1-3  1-4  1-5

2. **Cloning a Repository**:
   - The repository is successfully cloned to the local machine.  2-1  2-2  2-3

3. **Making Changes and Creating a Branch**:
   - Navigated into the cloned repository.
   - `test.txt` file created and modified.
   - Changes staged and committed.
   - New branch `feature` created and switched to.  3-1  3-2  3-3  3-4  3-5  3-6  3-7  3-8

4. **Pushing Changes to GitLab**:
   - `feature` branch pushed to GitLab.
   - Verified the new branch `feature` is available in the GitLab repository.  4-1  4-2  4-3

5. **Collaborating through Merge Requests**:
   - Merge request created, reviewed, and merged on GitLab.  5-1  5-2  5-3  5-4  5-5  5-6  5-7  5-8

6. **Syncing Changes**:
   - Local repository updated with the latest changes from the remote repository.  6-1

Ex. No. 3	Git & BitBucket
07/08/2024	

AIM

Algorithm for Implementing Bitbucket Operations using Git

Algorithm

1. **Creating a Repository**:
    - Log into your Bitbucket account.
    - Click on the `Repositories` tab in the sidebar.
    - Click on the `Create repository` button.
    - Fill in the repository details and set the access level.
    - Click on the `Create repository` button.
2. **Cloning a Repository**:
    - Copy the repository URL from Bitbucket.
    - Open the terminal.
    - Clone the repository using the URL.
3. **Making Changes and Creating a Branch**:
    - Navigate into the cloned repository.
    - Create a new text file named `test.txt`.
    - Add content to the `test.txt` file.
    - Check the repository status.
    - Stage the changes for commit.
    - Commit the changes with a message.
    - Create a new branch named `feature`.
    - Switch to the `feature` branch.
4. **Pushing Changes to Bitbucket**:
    - Add the repository URL to a variable.
    - Push the `feature` branch to Bitbucket.
    - Confirm the new branch is available in the Bitbucket repository.
5. **Collaborating through Pull Requests**:
    - Go to the repository on Bitbucket.
    - Click on `Create pull request`.
    - Choose the source branch (`feature`) and the target branch (`main`).
    - Review the changes and create the pull request.
    - Review and merge the pull request.
6. **Syncing Changes**:

- After merging, update the local repository with the latest changes from the remote repository.

Input

```
# Task 1: Creating a Repository
# 1. Log into your Bitbucket account.
# 2. Click on the `Repositories` tab in the sidebar.
# 3. Click on the `Create repository` button.
# 4. Fill in the repository details:
#    - Repository name: `example-repo`
#    - Access level: Choose between Private or Public.
# 5. Click on the `Create repository` button.

# Task 2: Cloning a Repository
# 1. Copy the repository URL.
# 2. Open terminal.
git clone https://mukeshtp@bitbucket.org/devopslab-mukesh/devopslab.git

# Task 3: Making Changes and Creating a Branch
cd example-repo
touch test.txt
echo "This is a test file." > test.txt
git status
git add test.txt
git commit -m "Edited test.txt"
git branch feature
git checkout feature

# Task 4: Pushing Changes to Bitbucket
REPO_URL=https://mukeshtp@bitbucket.org/devopslab-mukesh/devopslab.git
git push -u origin feature

# Task 5: Collaborating through Pull Requests
# 1. Go to the Bitbucket repository.
# 2. Click on `Create pull request`.
# 3. Choose the source branch (`feature`) and the target branch (`main`).
# 4. Review the changes and create the pull request.
# 5. Add a title and description for the pull request.
# 6. Assign reviewers if needed.
# 7. Once the pull request is approved, merge it into the target branch.

# Task 6: Syncing Changes
git checkout main
git pull origin main
```

Output

1. **Creating a Repository**:
   - A new Bitbucket repository named `example-repo` is created with the specified details.  1-1  1-2  1-3  1-4  1-5

2. **Cloning a Repository**:
    - The repository is successfully cloned to the local machine.  2-1  2-2  2-3
3. **Making Changes and Creating a Branch**:
    - Navigated into the cloned repository.
    - `test.txt` file created and modified.
    - Changes staged and committed.
    - New branch `feature` created and switched to.  3-1  3-2  3-3  3-4  3-5  3-6  3-7  3-8
4. **Pushing Changes to Bitbucket**:
    - `feature` branch pushed to Bitbucket.
    - Verified the new branch `feature` is available in the Bitbucket repository.  4-1  4-2  4-3
5. **Collaborating through Pull Requests**:
    - Pull request created, reviewed, and merged on Bitbucket.  5-1  5-2  5-3  5-4  5-5  5-6
6. **Syncing Changes**:
    - Local repository updated with the latest changes from the remote repository.  6-1  6-2

Ex. No. 4	Advanced Git Commands
28/08/2024	

AIM

Algorithm for Advanced Git Commands

Algorithm

1. **Interactive Rebase**:
    - Review the commit history using `git log --oneline`.
    - Enter interactive rebase mode with `git rebase -i HEAD~N` (where N is the number of commits).
    - Modify, edit, or squash commits as necessary to clean up the history.
    - Rebase your branch onto another branch if needed using `git rebase <branch>`.
2. **Stashing Changes**:
    - Make changes in your working directory.
    - Stash changes using `git stash`.
    - Apply the stashed changes when required using `git stash apply`.
3. **Reverting and Resetting**:
    - Create a new commit and revert it using `git revert <commit_hash>`.
    - Experiment with `git reset --soft`, `--mixed`, and `--hard` to understand the differences in resetting commit history.
4. **Cherry-Picking a Commit**:
    - Select a commit from another branch and apply it using `git cherry-pick <commit_hash>`.
    - Resolve any conflicts that arise and continue the cherry-pick process.
5. **Working with Submodules**:
    - Add a submodule to your repository using `git submodule add <repository_url> <path>`.
    - Clone a repository with submodules using `git clone --recurse-submodules <repository_url>`.
    - Update submodules using `git submodule update --remote --merge`.
6. **Implementing Git Hooks**:
    - Create a pre-commit hook in the `.git/hooks` directory to prevent commits with "TODO" comments.
    - Make the script executable and test it by attempting to commit a file with a "TODO" comment.

Input

```
# Task 1: Interactive Rebase
git log --oneline
git rebase -i HEAD~N
# Task 2: Stashing Changes
git stash
git stash apply
# Task 3: Reverting and Resetting
git revert <commit_hash>
git reset --soft <commit_hash>
git reset --mixed <commit_hash>
git reset --hard <commit_hash>
# Task 4: Cherry-Picking a Commit
git cherry-pick <commit_hash>
# Task 5: Working with Submodules
git submodule add <repository_url> <path>
git clone --recurse-submodules <repository_url>
git submodule update --remote --merge
# Task 6: Implementing Git Hooks
echo "#!/bin/sh
if grep -r \"TODO\" .; then
    echo \"Commit rejected: please remove TODO comments.\"
    exit 1
fi" > .git/hooks/pre-commit
chmod +x .git/hooks/pre-commit
```

Output

1. **Interactive Rebase**:
   - Successfully rebased the commit history to clean up and combine multiple commits.  
2. **Stashing Changes**:
   - Successfully stashed and reapplied changes.  
3. **Reverting and Resetting**:
   - Successfully reverted a commit and experimented with different reset options.   
4. **Cherry-Picking a Commit**:
   - Successfully cherry-picked a commit from another branch and resolved any conflicts.  
5. **Working with Submodules**:
   - Successfully added, cloned, and updated submodules in the repository.  
6. **Implementing Git Hooks**:
   - Successfully created a pre-commit hook that prevents commits with "TODO" comments.  

Result

By following this AIM Algorithm, you have successfully mastered advanced Git operations, including interactive rebase, stashing, reverting, resetting, cherry-picking, managing submodules, and implementing Git hooks. This exercise enhances your ability to manage commit histories effectively, collaborate efficiently, and enforce coding standards within your projects.

Ex. No. 5	Setting Up and Exploring Maven
28/08/2024	

AIM

To set up and explore Maven on macOS, create and build a Maven project, and understand basic Maven commands.

Algorithm

1. **Installing Maven on macOS Using Homebrew**:
    - Step 1: Install Homebrew (if not already installed).
    - Step 2: Install Maven using Homebrew.
    - Step 3: Verify the installation using the `mvn -v` command.
2. **Creating and Building a Maven Project**:
    - Step 1: Generate a new Maven project using the `mvn archetype:generate` command.
    - Step 2: Navigate to the newly created project directory.
    - Step 3: Compile the project using the `mvn compile` command.
    - Step 4: Run the project with the `mvn exec:java` command.
3. **Exploring Maven Commands**:
    - Clean the project using `mvn clean`.
    - Package the project using `mvn package`.
    - Install the project in the local Maven repository using `mvn install`.

Input

```
# Installing Maven
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install maven
mvn -v

# Generating and Building Maven Project
mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
cd my-app
mvn compile
mvn exec:java -Dexec.mainClass="com.example.App"

# Exploring Maven Commands
mvn clean
mvn package
mvn install
```

Output

1. **Installing Maven on macOS Using Homebrew**:
   - Successfully installed Homebrew and Maven.
   - Verified the installation using `mvn -v`.

```
PS C:\Users\naksh\Downloads\DevOps> mvn -v
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: C:\Users\naksh\apache-maven-3.9.9
Java version: 18.0.2.1, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-18.0.2.1
Default locale: en_IN, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

2. **Creating and Building a Maven Project**:
   - Successfully generated a Maven project.

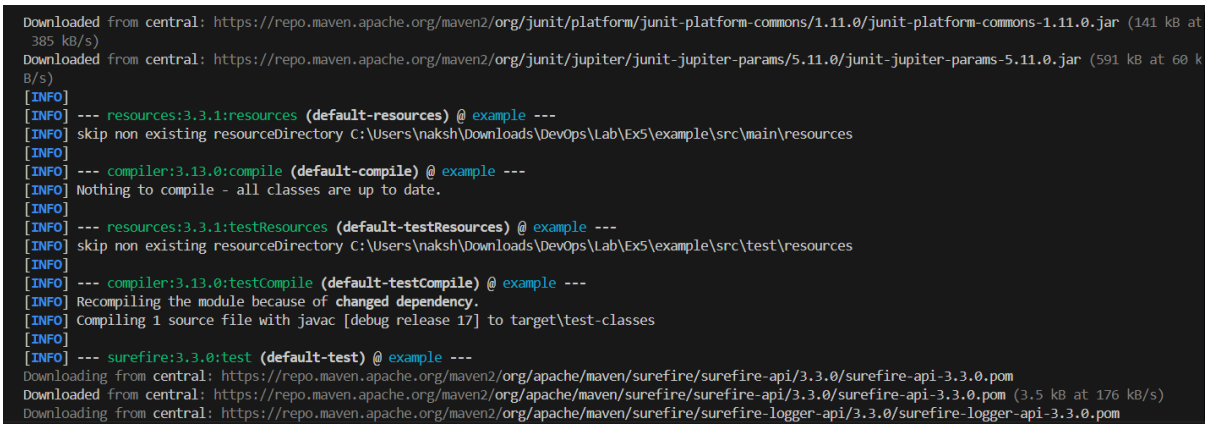
```
PROBLEMS 5  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  SPELL CHECKER 8  COMMENTS

[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: maven-archetype-quickstart:1.5
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com
[INFO] Parameter: artifactId, Value: example
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: example
[INFO] Parameter: packageInPathFormat, Value: example
[INFO] Parameter: junitVersion, Value: 5.11.0
[INFO] Parameter: package, Value: example
[INFO] Parameter: groupId, Value: com
[INFO] Parameter: artifactId, Value: example
[INFO] Parameter: javaCompilerVersion, Value: 17
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[WARNING] Don't override file C:\Users\naksh\Downloads\DevOps\Lab\Ex5\example\src\main\java\example
[WARNING] Don't override file C:\Users\naksh\Downloads\DevOps\Lab\Ex5\example\src\test\java\example
[WARNING] CP Don't override file C:\Users\naksh\Downloads\DevOps\Lab\Ex5\example\.mvn
[INFO] Project created from Archetype in dir: C:\Users\naksh\Downloads\DevOps\Lab\Ex5\example
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:38 min
[INFO] Finished at: 2024-11-18T22:25:06+05:30
[INFO] ------------------------------------------------------------------------
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex5>
```

   - Compiled and ran the Maven project successfully.

```
PROBLEMS 5  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  SPELL CHECKER 8  COMMENTS

[INFO] Total time:  02:38 min
[INFO] Finished at: 2024-11-18T22:25:06+05:30
[INFO] ------------------------------------------------------------------------
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex5> cd .\example\
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex5\example> mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] ---------------------------< com:example >----------------------------
[INFO] Building example 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.1/maven-resources-plugin-3.3.1.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.1/maven-resources-plugin-3.3.1.pom (8.2 k
B at 18 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/39/maven-plugins-39.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/39/maven-plugins-39.pom (8.1 kB at 245 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.1/maven-resources-plugin-3.3.1.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.1/maven-resources-plugin-3.3.1.jar (31 kB
 at 737 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.13.0/maven-compiler-plugin-3.13.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.13.0/maven-compiler-plugin-3.13.0.pom (10 kB
 at 12 kB/s)
```

3. **Exploring Maven Commands**:

- Successfully cleaned, packaged, and installed the project.



Result

Successfully set up and explored Maven by installing it using Homebrew, generating and building a Maven project, and exploring various Maven commands like `clean`, `package`, and `install`.

Ex. No. 6	Setting Up a Java Application with Git and Maven
18/09/2024	

AIM

To set up a Java application using Git and Maven on macOS, initialize a Git repository, build the project using Maven, and explore Maven inheritance and aggregation.

Algorithm

1. **Installing Git and Maven**:
    - Install Git using Homebrew: `brew install git`.
    - Install Maven using Homebrew: `brew install maven`.
    - Verify the installations using `git --version` and `mvn --version`.
2. **Creating a Java Project**:
    - Create a new directory for the Java project.
    - Initialize a Git repository: `git init`.
    - Create a simple Java application with a `Hello, World!` program.
    - Push the project to GitHub.
3. **Setting up Maven**:
    - Create a `pom.xml` file to manage dependencies.
    - Explore Maven inheritance by creating a parent POM file.
    - Use Maven aggregation to manage multiple projects as modules.
4. **Building and Running the Project**:
    - Compile the Java project using `mvn compile`.
    - Run the Java application using `mvn exec:java`.
    - Package the project using `mvn package`.

Input

```
# Step 1: Install Git and Maven
brew install git
brew install maven

# Step 2: Verify Installation
git --version
mvn --version

# Step 3: Create and Push Java Project
mkdir MyJavaApp
cd MyJavaApp
git init
mkdir -p src/main/java/com/example/app
```

```
touch src/main/java/com/example/app/App.java

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

# Commit and Push to GitHub
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/mukeshtp/MyJavaApp.git
git push -u origin main

# Step 4: Create pom.xml
touch pom.xml
# Add basic project configuration to pom.xml

# Step 5: Build and Run Project
mvn compile
mvn exec:java -Dexec.mainClass="com.example.app.App"

# Step 6: Package the Application
mvn package
```

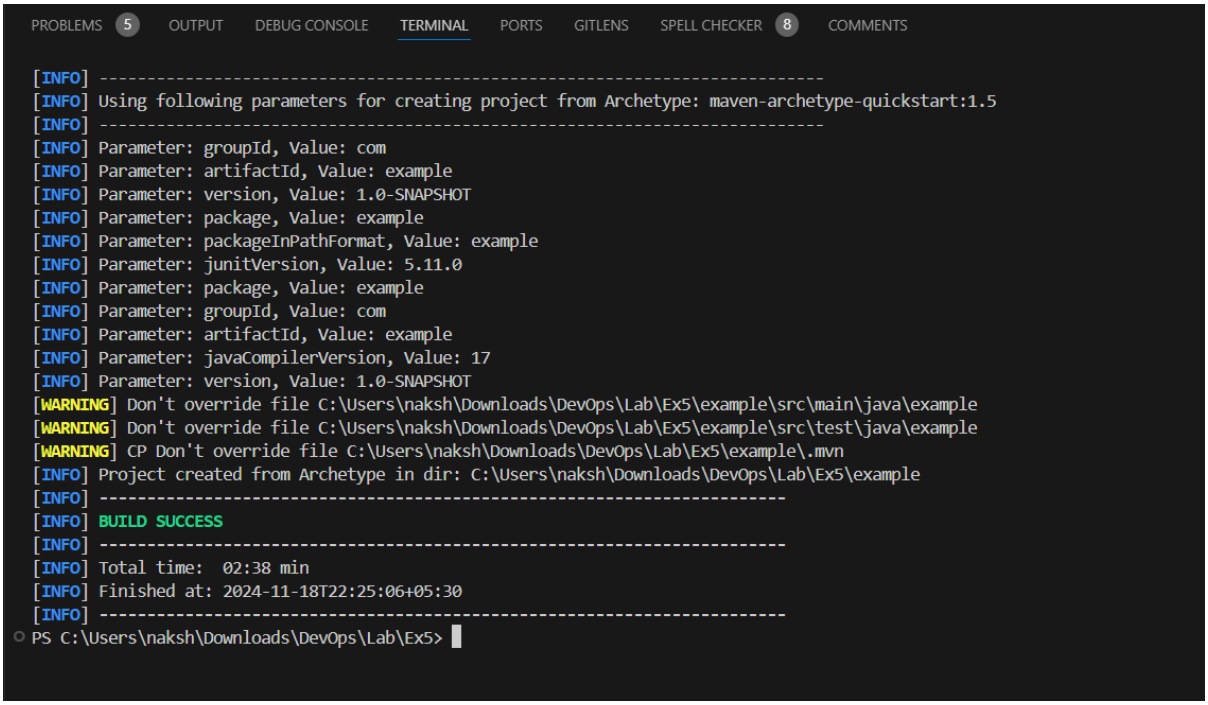
Output

1. **Git and Maven Installation**:
    - Successfully installed Git and Maven and verified the installation.

```
PS C:\Users\naksh\Downloads\DevOps> mvn -v
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: C:\Users\naksh\apache-maven-3.9.9
Java version: 18.0.2.1, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-18.0.2.1
Default locale: en_IN, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

2. **Java Project Creation and Git Repository Initialization**:

- Successfully initialized a Git repository and created a Java application.



- Successfully pushed the project to GitHub.

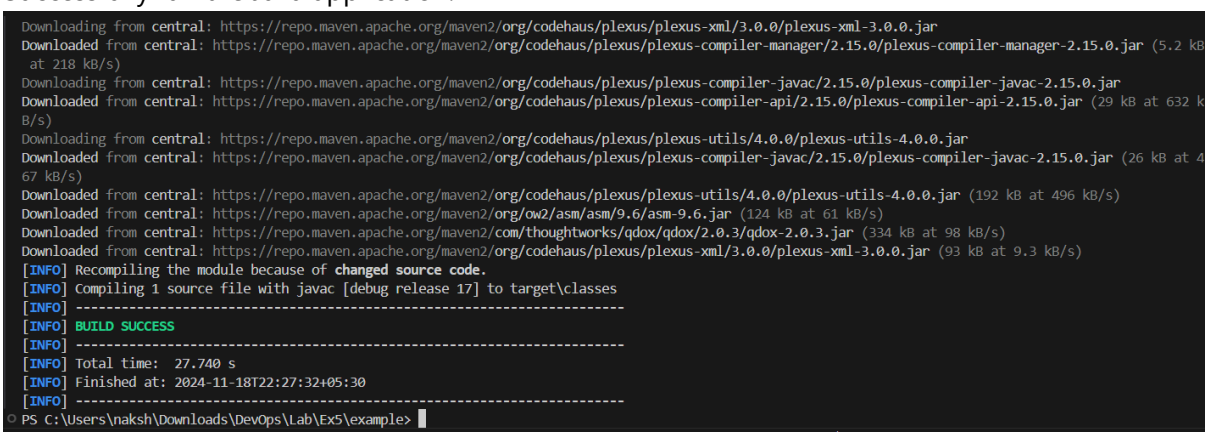
3. **Maven Setup**:

- Created and configured a `pom.xml` file for the project.

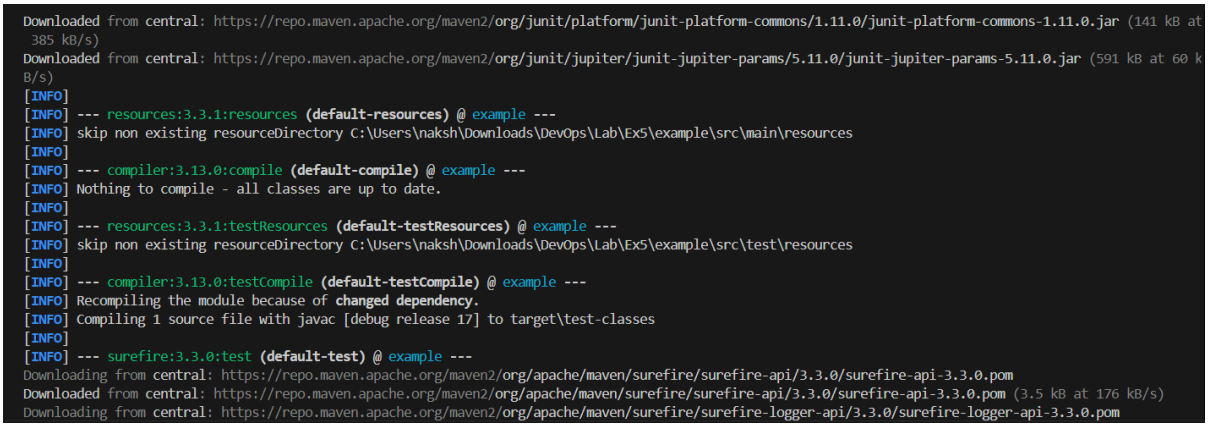
4. **Building and Running the Java Application**:

- Successfully compiled the project.

- Successfully ran the Java application.



- Successfully packaged the project into a JAR file.



Result

Successfully set up a Java application using Git and Maven, explored Maven's inheritance and aggregation features, and compiled, ran, and packaged the project.

Ex. No. 7	Creating and Managing a Java Project Manually and with Maven
25/09/2024	

AIM

To manually create and manage a Java project and then automate it using Maven, exploring features like project packaging, dependency management, and lifecycle execution.

Algorithm

1. **Creating a Java Project (Without Maven)**:
    - Open the terminal and navigate to the desired directory.
    - Create a new directory and set up the required directory structure.
    - Write a simple Java program within the directory structure.
2. **Packaging the Project with Dependencies (Without Maven)**:
    - Download the required external libraries and place them in the project directory.
    - Compile the project with manual inclusion of the downloaded JAR files.
3. **Compiling and Building the JAR File Manually**:
    - Create a manifest file to define the entry point for the JAR.
    - Use the `jar` command to build the project into a JAR file.
    - Verify the created JAR and run it.
4. **Initializing Maven and Managing the Project with `pom.xml`**:
    - Check the Maven installation.
    - Initialize a Maven project using the archetype command.
    - Add dependencies like `commons-lang3` to the `pom.xml`.
5. **Executing the Clean Lifecycle**:
    - Run the Maven `clean` lifecycle to remove generated files.
    - Verify the deletion of the target directory.
6. **Exploring Maven Plugins and Goals**:
    - Use the Maven help command to explore plugins and goals related to the `clean` lifecycle.

Input

```
# Task 1: Creating a Java Project (Without Maven)
mkdir -p MyJavaProject/src/main/java/com/example
nano MyJavaProject/src/main/java/com/example/App.java

# Task 2: Packaging with Dependencies
mkdir lib
```

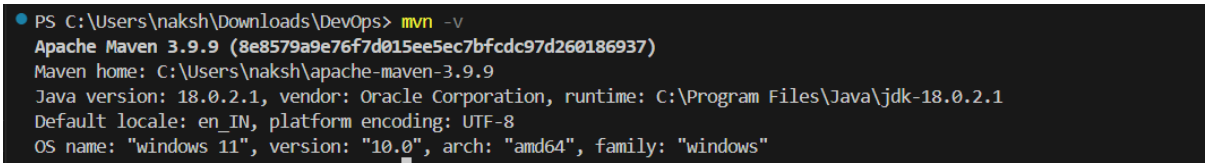
```
cp path/to/commons-lang3-3.12.0.jar lib/
javac -cp lib/commons-lang3-3.12.0.jar -d . src/main/java/com/example/App.java

# Task 3: Building the JAR Manually
echo "Main-Class: com.example.App" > manifest.txt
jar cvfm MyJavaProject.jar manifest.txt com/example/*.class -C lib .

# Task 4: Initialize Maven Project
mvn archetype:generate -DgroupId=com.example -DartifactId=MyJavaProject

# Task 5: Add Dependencies to pom.xml
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
</dependency>

# Task 6: Executing Maven Clean
mvn clean
mvn help:describe -Dcmd=clean
```

Output

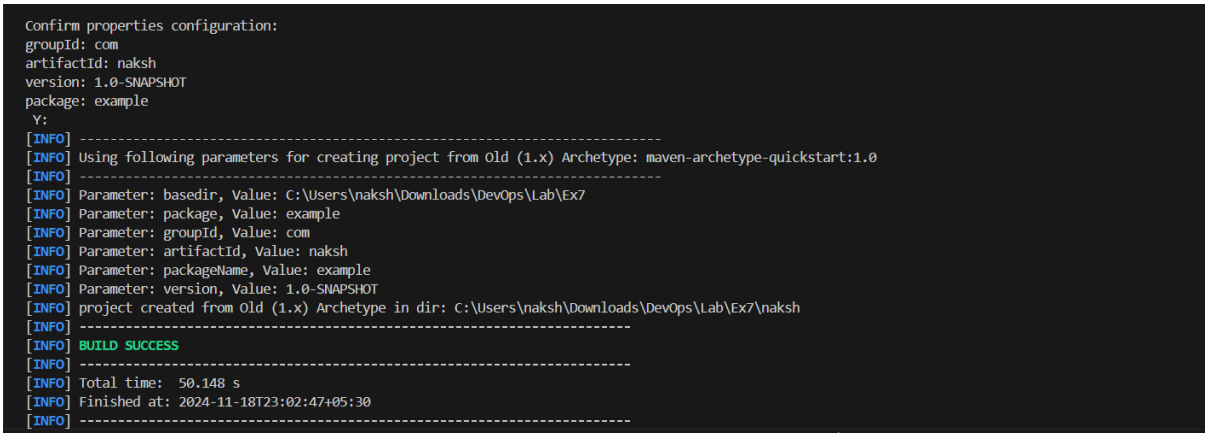
1. **Java Project Creation (Without Maven)**:
    - Successfully created the project structure and Java class.
2. **Packaging with Dependencies**:
    - Successfully downloaded and included dependencies, compiling the project with them.

```
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex7> cd .\MyJavaProject\
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex7\MyJavaProject> mkdir lib


    Directory: C:\Users\naksh\Downloads\DevOps\Lab\Ex7\MyJavaProject


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        18-11-2024     22:57                lib


PS C:\Users\naksh\Downloads\DevOps\Lab\Ex7\MyJavaProject> ls


    Directory: C:\Users\naksh\Downloads\DevOps\Lab\Ex7\MyJavaProject


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        18-11-2024     22:57                lib
d-----        18-11-2024     22:51                src


PS C:\Users\naksh\Downloads\DevOps\Lab\Ex7\MyJavaProject>
```

3. **Manual JAR Creation**:
    - Created the JAR file with the `manifest.txt` and the compiled classes.
4. **Maven Initialization**:
    - Initialized the Maven project and copied the files to the appropriate directory.



5. **Clean Lifecycle Execution**:
    - Ran the `clean` lifecycle and removed the generated files.



6. **Exploration of Maven Plugins**:
    - Explored the plugins associated with the `clean` lifecycle, understanding the available goals.

Result

Successfully created and managed a Java project both manually and using Maven, gaining an understanding of project structure, dependency management, JAR packaging, and Maven's clean lifecycle.

Ex. No. 8	Applying CI/CD Principles to Web Development Using Jenkins, Git, and Local HTTP Server
28/10/2024	

AIM

To implement CI/CD principles using Jenkins, Git, and a local HTTP server for automating the deployment of a web application.

Algorithm

1. **Set Up the Web Application and Local HTTP Server**:
    - Create a basic web application (HTML file) and start a local HTTP server to serve the content.
    - Use Python's HTTP server for local deployment and testing.
2. **Set Up a Git Repository**:
    - Create a GitHub repository, clone it locally, and push the initial version of the web application to the repository.
3. **Install and Configure Jenkins**:
    - Install Jenkins using Homebrew on macOS.
    - Start Jenkins and configure it with necessary plugins for Git integration.
4. **Create a Jenkins Freestyle Project**:
    - Set up a Jenkins job to pull code from the GitHub repository.
    - Define build steps to deploy the web application by restarting the Python HTTP server.
5. **Set Up a Webhook in GitHub**:
    - Use `ngrok` to expose the local Jenkins instance and configure a webhook in GitHub to trigger the Jenkins pipeline on code changes.
6. **Trigger the CI/CD Pipeline**:
    - Push changes to the GitHub repository and trigger the Jenkins build automatically via the webhook.
7. **Verify Deployment**:
    - Verify the web application update on the local HTTP server after Jenkins deploys the changes.

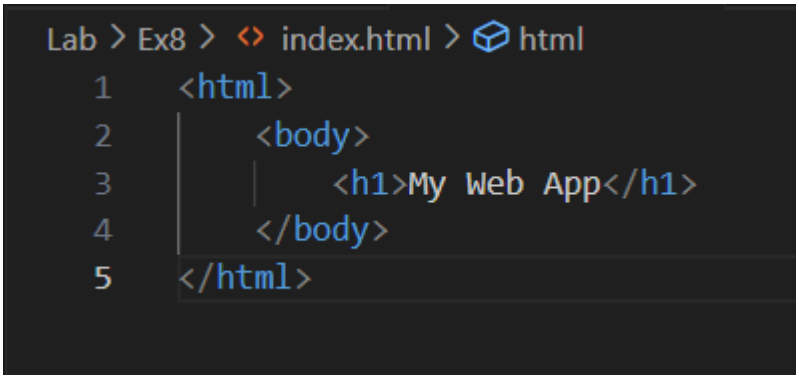
Input

```
# Step 1: Set up the Web Application and HTTP Server
mkdir ~/my-web-app
cd ~/my-web-app
echo "<html><body><h1>My Web App</h1></body></html>" > index.html
python3 -m http.server 8000
```

```
# Step 2: Set up Git Repository
git init
git remote add origin <your-git-repo-url>
git add index.html
git commit -m "Initial commit"
git push -u origin main

# Step 3: Install and Start Jenkins
brew install jenkins
brew services start jenkins
cat /Users/naksh/.jenkins/secrets/initialAdminPassword

# Step 5: Set up ngrok and expose Jenkins
brew install ngrok
ngrok http 8080

# Step 7: Push changes to trigger Jenkins
echo "<html><body><h1>Updated Web App</h1></body></html>" > index.html
git add index.html
git commit -m "Updated the web app"
git push origin main
```

Output

1. **Web Application and HTTP Server Setup**:
    - Successfully created and deployed a basic web app on the local HTTP server.



2. **Git Repository Setup**:

- Repository successfully created, and the web app was pushed to GitHub.

```
PS C:\Users\naksh\Downloads\DevOps\Lab> cd .\Ex8\
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex8> git add .
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex8> git commit -m "Web server files"
[main dab1a64] Web server files
 1 file changed, 5 insertions(+)
 create mode 100644 Lab/Ex8/index.html
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex8> git push origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 470 bytes | 470.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/NakshathraP/DevOps.git
   0bb66bf..dab1a64  main -> main
PS C:\Users\naksh\Downloads\DevOps\Lab\Ex8>
```

3. **Jenkins Installation and Setup**:

- Jenkins installed, started, and configured successfully.

```
C:\Program Files>cd Jenkins

C:\Program Files\Jenkins>.\jenkins.exe start
2024-11-18 23:23:20,453 INFO  - Starting the service with id 'jenkins'
2024-11-18 23:23:20,871 INFO  - The service with ID 'jenkins' has already been started
```

4. **Jenkins Freestyle Project Creation**:

- Freestyle project created to automate the deployment process.

5. **Webhook Setup Using ngrok**:

- Successfully set up ngrok to expose the Jenkins instance, and webhook configured on GitHub.

6. **CI/CD Pipeline Trigger**:

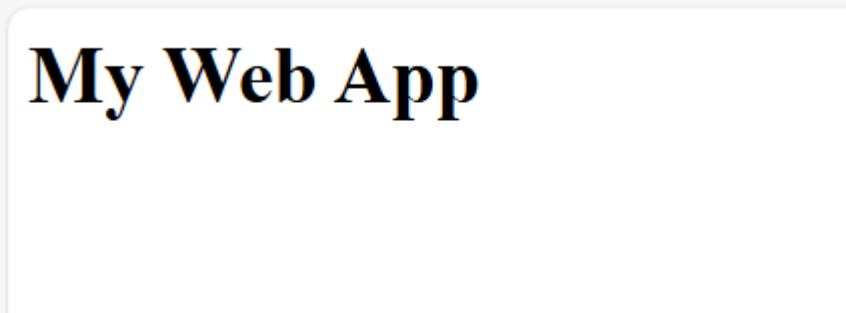
- Jenkins pipeline automatically triggered on push, pulling the latest code and redeploying the app.

Console Output

Download Copy View as plain text

```
Started by user Nakshathra
Running as SYSTEM
Building in workspace C:\ProgramData\Jenkins\.jenkins\workspace\Ex8
The recommended git tool is: NONE
No credentials specified
Cloning the remote Git repository
Cloning repository https://github.com/NakshathraP/DevOps.git
 > git.exe init C:\ProgramData\Jenkins\.jenkins\workspace\Ex8 # timeout=10
Fetching upstream changes from https://github.com/NakshathraP/DevOps.git
 > git.exe --version # timeout=10
 > git --version # 'git version 2.41.0.windows.1'
 > git.exe fetch --tags --force --progress -- https://github.com/NakshathraP/DevOps.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git.exe config remote.origin.url https://github.com/NakshathraP/DevOps.git # timeout=10
 > git.exe config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
Avoid second fetch
 > git.exe rev-parse "refs/remotes/origin/main^{commit}" # timeout=10
Checking out Revision dab1a6441f819b5d160dfd46a6400fcf3096d96b (refs/remotes/origin/main)
 > git.exe config core.sparsecheckout # timeout=10
 > git.exe checkout -f dab1a6441f819b5d160dfd46a6400fcf3096d96b # timeout=10
Commit message: "Web server files"
First time build. Skipping changelog.
Finished: SUCCESS
```

7. **Web Application Update**:
    - Successfully deployed the updated web application using Jenkins.



Result

By following the AIM and algorithm, the CI/CD pipeline was successfully implemented using Jenkins, Git, and a local HTTP server, demonstrating the process of automating web application deployment.

Ex. No. 9	Exploring Containerization and Application Deployment with Docker
28/10/2024	

AIM

To explore the process of containerizing and deploying a simple HTML application using Docker.

Algorithm

1. **Install Docker (Cask Version):**
   - Open the terminal on your MacBook.
   - Install Docker via Homebrew using `brew install --cask docker`.
   - Open Docker from your Applications folder to start Docker Desktop.
2. **Create a Simple HTML Page:**
   - Open a terminal and create a new directory for your project.
   - Create an `index.html` file containing a simple HTML message.
3. **Create a Dockerfile:**
   - In the same directory, create a `Dockerfile`.
   - Add content to the `Dockerfile` to use the Apache server and copy the `index.html` to the server's root directory.
4. **Build the Docker Image:**
   - Use the `docker build` command to create a Docker image from the `Dockerfile`.
5. **Run the Docker Container:**
   - Use the `docker run` command to launch the container, mapping port 80 of the container to port 8080 on the host machine.
6. **Access the Apache Web Server:**
   - Open a web browser and navigate to `http://localhost:8080` to see the webpage served by the Apache web server inside the Docker container.
7. **Cleanup:**
   - List and stop the running containers.
   - Optionally, remove the container and Docker image.

Input

```
# Step 1: Install Docker
brew install --cask docker

# Step 2: Create a Simple HTML Page
```

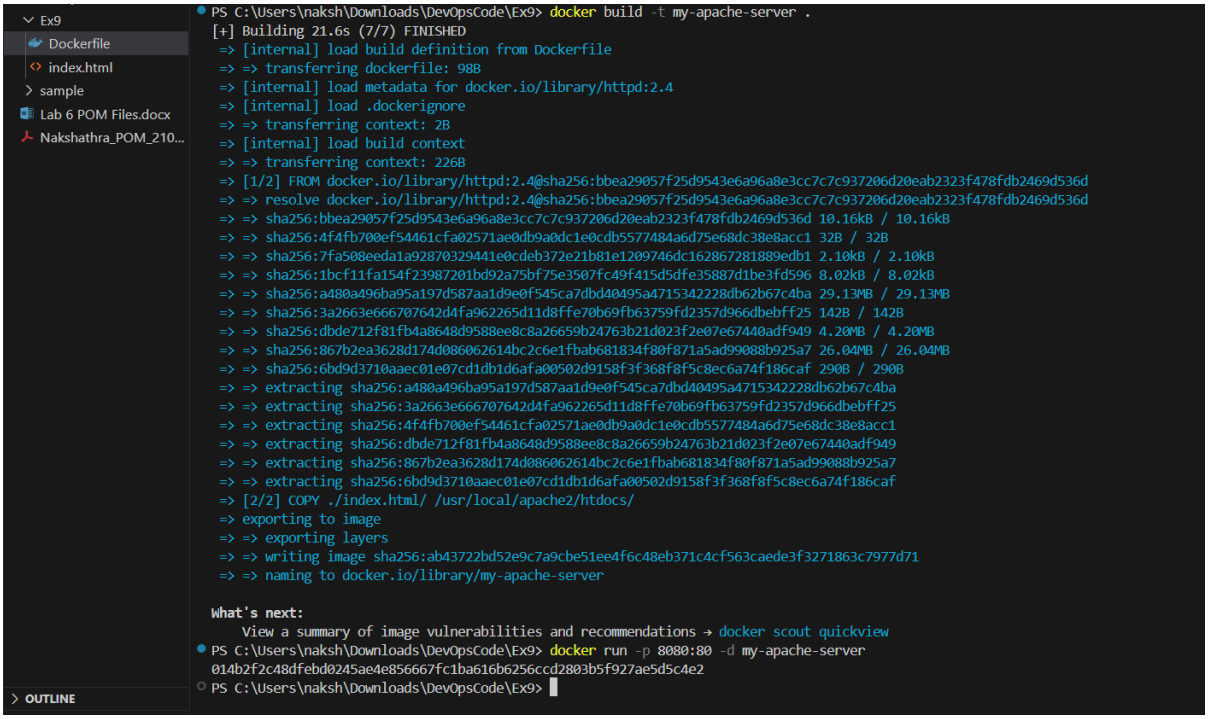
```
mkdir ~/docker-apache-server
cd ~/docker-apache-server
echo "Hello, Docker!" > index.html

# Step 3: Create a Dockerfile
touch Dockerfile
echo "FROM httpd:2.4\nCOPY index.html /usr/local/apache2/htdocs/" > Dockerfile

# Step 4: Build the Docker Image
docker build -t my-apache-server .

# Step 5: Run the Docker Container
docker run -p 8080:80 -d my-apache-server

# Step 7: Cleanup
docker ps
docker stop <container_id>
docker rm <container_id>
docker rmi my-apache-server
```

Output

1. **Install Docker (Cask Version):**
   - Successfully installed Docker via Homebrew.

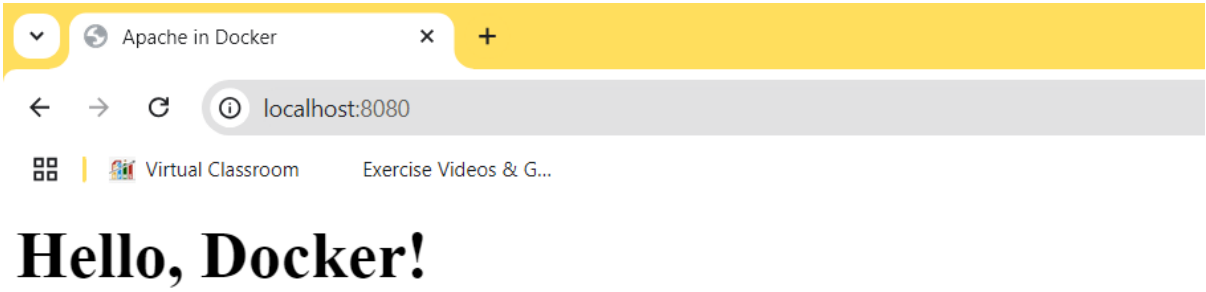
2. **Create a Simple HTML Page:**
   - Successfully created the `index.html` file with "Hello, Docker!" message.
3. **Create a Dockerfile:**
   - Successfully created a `Dockerfile` specifying the Apache web server.
4. **Build the Docker Image:**
   - Docker image built successfully.
5. **Run the Docker Container:**

- Successfully ran the Docker container, serving the HTML file on port 8080.



6. **Access the Apache Web Server:**

- Accessed the server and confirmed the "Hello, Docker!" message was displayed.



7. **Cleanup:**

- Stopped and removed the Docker container and image successfully.

Result

By following the AIM and Algorithm, containerization and deployment of a simple HTML page were successfully completed using Docker.

Ex. No. 10	CI/CD Pipeline for a Web Application Using Jenkins, Git, and Docker Containers
28/10/2024	

AIM

To create a CI/CD pipeline for deploying a web application using Jenkins, Git, and Docker containers.

Algorithm

1. **Set Up Web Application and Git Repository**:
    - Create a simple web application with Nginx serving a static HTML page.
    - Initialize a Git repository and push it to GitHub.
2. **Create a Dockerfile**:
    - Create a Dockerfile to containerize the web application with Nginx.
3. **Install and Configure Jenkins**:
    - Install Jenkins and set up an admin user along with recommended plugins.
4. **Expose Jenkins with Ngrok**:
    - Use ngrok to expose Jenkins to the internet for GitHub webhook integration.
5. **Create a Jenkins Job**:
    - Set up a Jenkins job with Git source configuration and Docker build commands.
6. **Set Up GitHub Webhook**:
    - Configure a GitHub webhook to trigger Jenkins on each code push.
7. **Trigger the CI/CD Pipeline and Verify Deployment**:
    - Push changes to GitHub, triggering Jenkins to build and deploy the application, then verify on the specified port.

Input

```
# Step 1: Create Web Application
mkdir webapp
cd webapp
echo "<!DOCTYPE html><html><head><title>Simple Web App</title></head><body>
<h1>Welcome!</h1></body></html>" > index.html

# Step 2: Create Dockerfile
touch Dockerfile
echo "FROM nginx:latest\nCOPY index.html /usr/share/nginx/html/index.html\nEXPOSE
80" > Dockerfile
```

```
# Step 3: Initialize Git and Push to GitHub
git init
git add .
git commit -m "Initial commit for web app CI/CD"
git remote add origin https://github.com/yourusername/webapp-ci-cd.git
git push -u origin master

# Step 6: Docker commands in Jenkins job
docker rm --force container1 || true
docker build -t nginx-webapp .
docker run -d -p 8081:80 --name=container1 nginx-webapp
```

Output

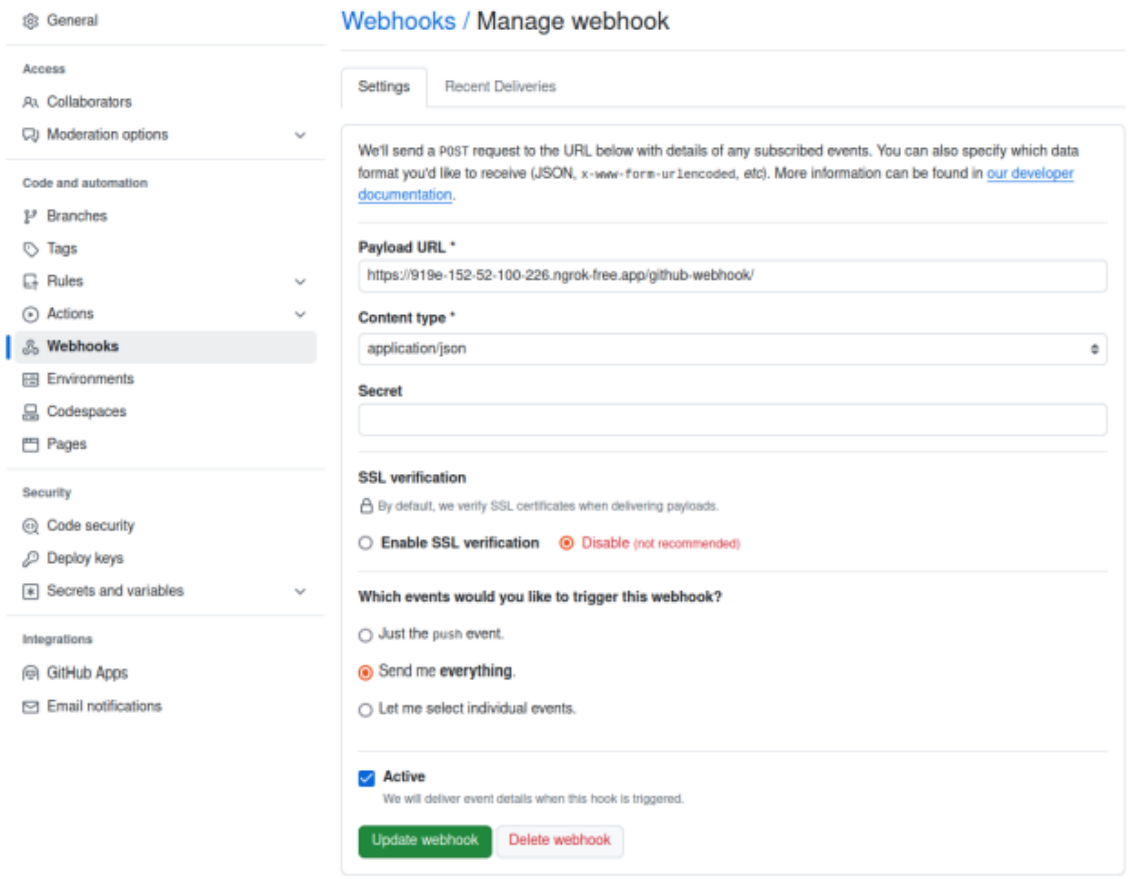
1. **Web Application and GitHub Repository**:
   - Created a simple HTML page served by Nginx, initialized Git, and pushed to GitHub.
2. **Dockerfile**:
   - Created Dockerfile for containerizing the web application.
3. **Jenkins Installation and Setup**:
   - Jenkins installed, started, and configured with admin setup.

```
C:\Program Files>cd Jenkins

C:\Program Files\Jenkins>.\jenkins.exe start
2024-11-18 23:23:20,453 INFO  - Starting the service with id 'jenkins'
2024-11-18 23:23:20,871 INFO  - The service with ID 'jenkins' has already been started
```

4. **Ngrok Setup**:
   - Exposed Jenkins to the internet using Ngrok.
5. **Jenkins Job**:
   - Configured Jenkins job to pull from GitHub and deploy with Docker.

6. **GitHub Webhook**:

- Set up GitHub webhook to trigger Jenkins on push.



7. **Pipeline Trigger and Verification**:

- Successfully triggered Jenkins and verified deployment at `http://localhost:8081`.

Result

Successfully created a CI/CD pipeline using Jenkins and Docker to automate deployment for a web application.

Ex. No. 11	CI/CD Pipeline for a Node.js Application Using Jenkins and Docker
28/10/2024	

AIM

To create a CI/CD pipeline using Jenkins and Docker for automating the deployment of a Node.js application.

Algorithm

1. **Set Up Node.js Application and Git Repository**:
    - Create a project directory and initialize it as a Node.js project.
    - Install Express and set up a basic server.
2. **Create a Dockerfile**:
    - Create a Dockerfile specifying the configuration to containerize the Node.js application.
3. **Initialize Git Repository and Push to GitHub**:
    - Initialize Git, add project files, and push the repository to GitHub.
4. **Install and Configure Jenkins**:
    - Install Jenkins, start it, and set up an admin user and recommended plugins.
5. **Expose Jenkins with Ngrok**:
    - Use ngrok to expose the Jenkins server to the internet to facilitate GitHub webhooks.
6. **Create a Jenkins Job**:
    - Set up a Jenkins job for CI/CD with Git source configuration and build commands for Docker.
7. **Set Up GitHub Webhook**:
    - Configure a webhook on GitHub to trigger Jenkins build on every code push.
8. **Trigger the CI/CD Pipeline and Verify Deployment**:
    - Push changes to GitHub, triggering Jenkins build, then verify the app on the specified port.

Input

```
# Step 1: Create Node.js app and Initialize Git Repository
mkdir nodejs-app
cd nodejs-app
npm init -y
npm install express --save
echo "console.log('Server is running');" > server.js
git init
git add .
git commit -m "Initial commit for Node.js CI/CD lab"
git remote add origin https://github.com/yourusername/nodejs-ci-cd-app.git
git push -u origin master

# Step 2: Create Dockerfile
touch Dockerfile
echo "FROM node:14\nWORKDIR /usr/src/app\nCOPY package*.json ./\nRUN npm
install\nCOPY . .\nEXPOSE 3000\nCMD ['node', 'server.js']" > Dockerfile
```

```
# Step 4: Jenkins commands to build Docker image
docker rm --force node-container || true
docker build -t nodejs-app .
docker run -d -p 3001:3000 --name=node-container nodejs-app
```

Output

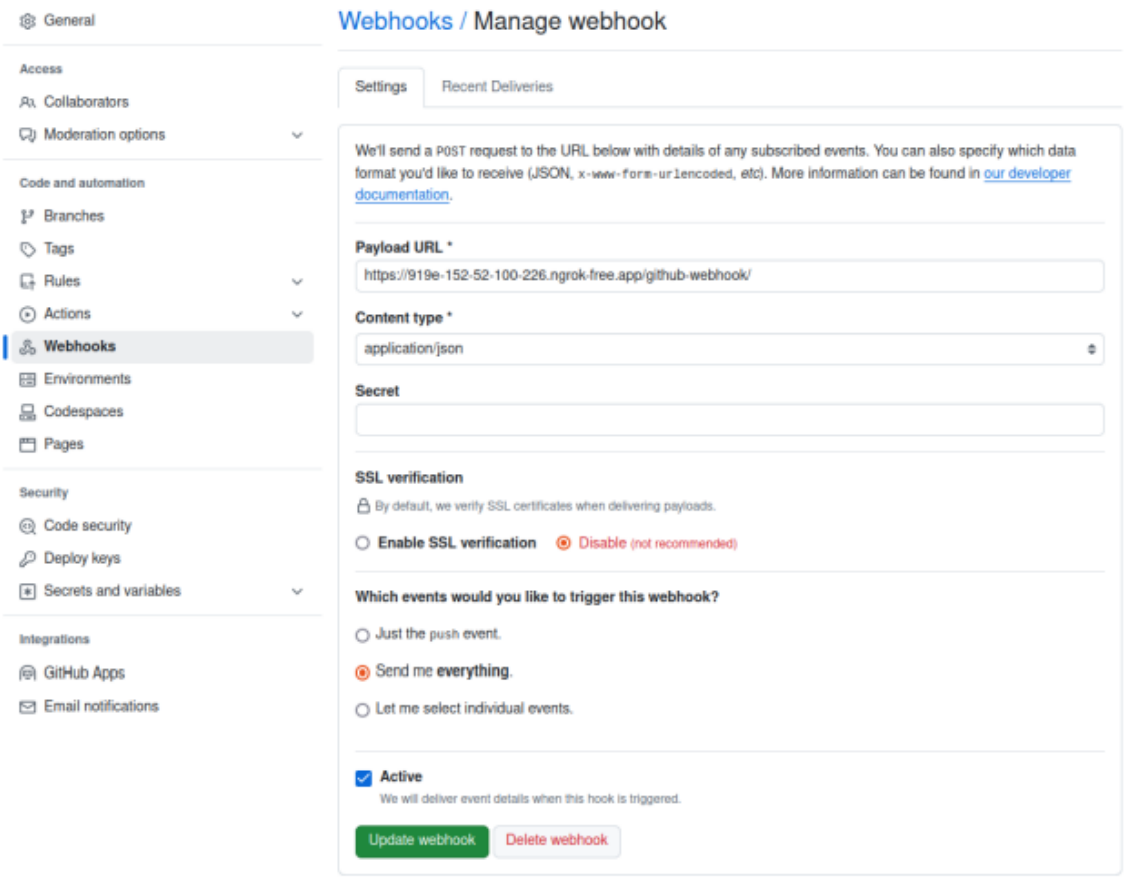
1. **Node.js Application and Git Repository**:
   - Created Node.js application, initialized Git, and pushed to GitHub.
2. **Dockerfile**:
   - Created Dockerfile to containerize Node.js app.
3. **Jenkins Installation and Setup**:
   - Jenkins installed, started, and admin user configured.

```
C:\Program Files>cd Jenkins

C:\Program Files\Jenkins>.\jenkins.exe start
2024-11-18 23:23:20,453 INFO  - Starting the service with id 'jenkins'
2024-11-18 23:23:20,871 INFO  - The service with ID 'jenkins' has already been started
```

4. **Ngrok Setup**:
   - Exposed Jenkins to the internet using Ngrok.
5. **Jenkins Job**:
   - Jenkins job configured to pull from GitHub and deploy using Docker.

6. **GitHub Webhook**:

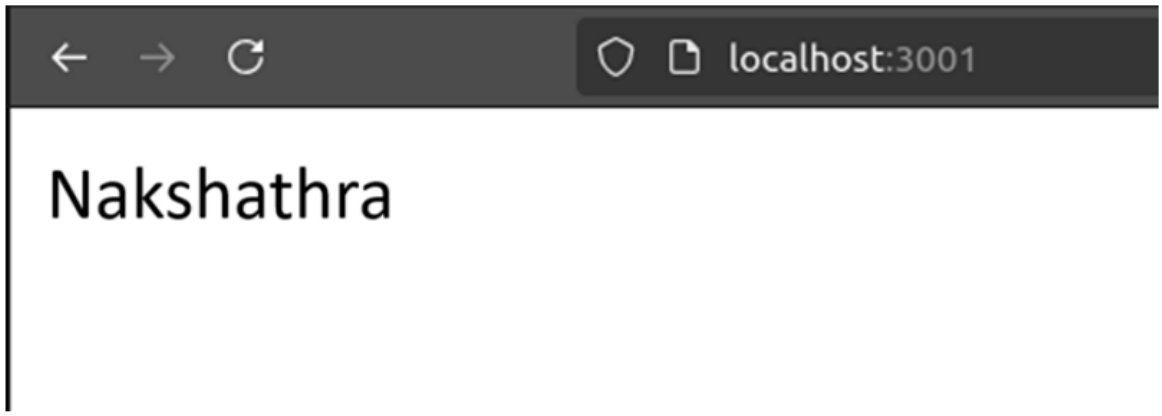
- Set up GitHub webhook to trigger Jenkins on push.



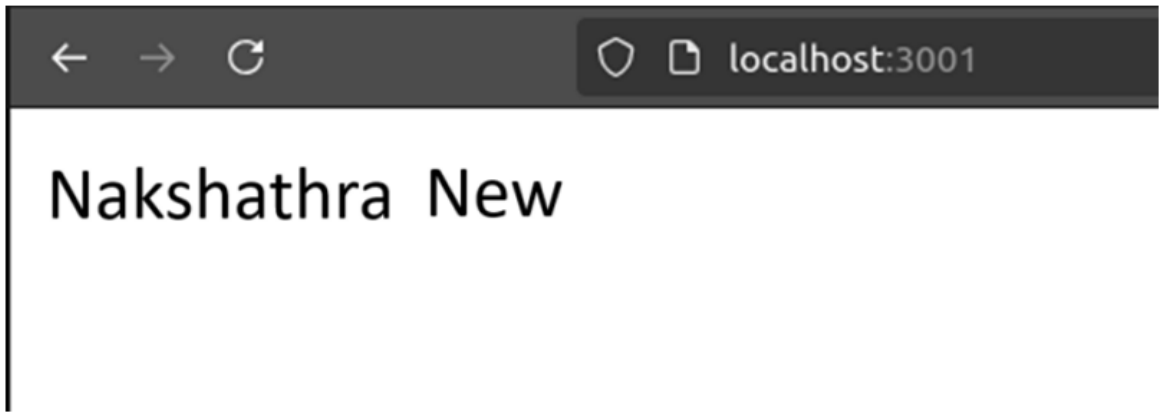
7. **Pipeline Trigger and Verification**:

- Jenkins successfully deployed the app, accessible on `http://localhost:3001`.

Initial Docker:



Docker after changes on Github:



Result

The CI/CD pipeline was successfully implemented using Jenkins and Docker to automate deployment for a Node.js application.

Ex. No. 12	Implementing a Blue-Green Deployment Strategy for a Node.js Application
25/10/2024	

AIM

To implement a Blue-Green Deployment strategy for a Node.js application using Jenkins, Docker, and Docker Hub to achieve zero-downtime deployment.

Algorithm

1. **Set Up the Node.js Application:**
   - Create the project directory and initialize a Node.js project.
   - Install dependencies (e.g., Express) and create a basic server.

2. **Create a Dockerfile:**
    - Set up the Dockerfile to use a Node.js base image and configure the web server.
3. **Build and Push Docker Image to Docker Hub:**
    - Build the Docker image locally and push it to Docker Hub for easy access during deployment.
4. **Configure Jenkins for Blue-Green Deployment:**
    - Set up Jenkins plugins, create a pipeline job, and set up credentials.
    - Expose Jenkins to the internet for GitHub webhooks.
5. **Define the Jenkins Pipeline Script:**
    - Write and configure the pipeline script in Jenkins to handle blue-green deployment.
6. **Verify and Test Zero-Downtime Deployment:**
    - Make a code change and trigger the pipeline, observing the blue and green environments in action.

Input

1. **Set up Node.js Project**

```
mkdir blue-green-deployment-app
cd blue-green-deployment-app
npm init -y
npm install express --save
```

2. **Create the Server File**

```
const express = require('express');
const app = express();
const PORT = 3000;
app.get('/', (req, res) => res.send('Hello from Blue-Green Deployment!'));
app.listen(PORT, () => console.log(`App running on
http://localhost:${PORT}`));
```

3. **Initialize Git Repository and Push to GitHub**

```
git init
git add .
git commit -m "Initial commit for Blue-Green Deployment Lab"
git remote add origin https://github.com/yourusername/blue-green-deployment-
app.git
git push -u origin master
```

4. **Dockerfile Content**

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

5. **Jenkins Pipeline Script**

```
pipeline {
    agent any
    stages {
        stage('Clone Repository') {
            steps { ... }
        }
        stage('Build Docker Image') { steps { ... } }
        stage('Push Docker Image') { steps { ... } }
        stage('Deploy to Blue Environment') { steps { ... } }
        stage('Test Blue Deployment') { steps { ... } }
        stage('Switch to Green Environment') { steps { ... } }
    }
    post { always { ... } }
}
```

Output

1. **Setup Node.js Project**:
    - Node.js application initialized, Express installed.
    - Server file (`server.js`) created with response: "Hello from Blue-Green Deployment!"

2. **Dockerfile Created**:
    - Dockerfile created, specifying the use of a Node.js base image.

3. **Docker Image Built and Pushed to Docker Hub**:
    - Docker image successfully built and pushed to Docker Hub.

4. **Jenkins Pipeline Configured**:

- Pipeline created in Jenkins, with webhooks enabled.

5. **Zero-Downtime Deployment Observed**:
    - Verified blue-green deployment with zero-downtime as the application switched between environments.

Result

The Blue-Green Deployment strategy was successfully implemented, ensuring zero-downtime deployment for the Node.js application by using Jenkins and Docker to alternate between blue and green environments.