|  |  |
| --- | --- |
| **Ex. No. 1** | Git and GitHub Basics |
|  |

**Aim:**

To explore Git commands through collaborative coding and Implement GitHub Operations using Git.

**Algorithm:**

Git Commands

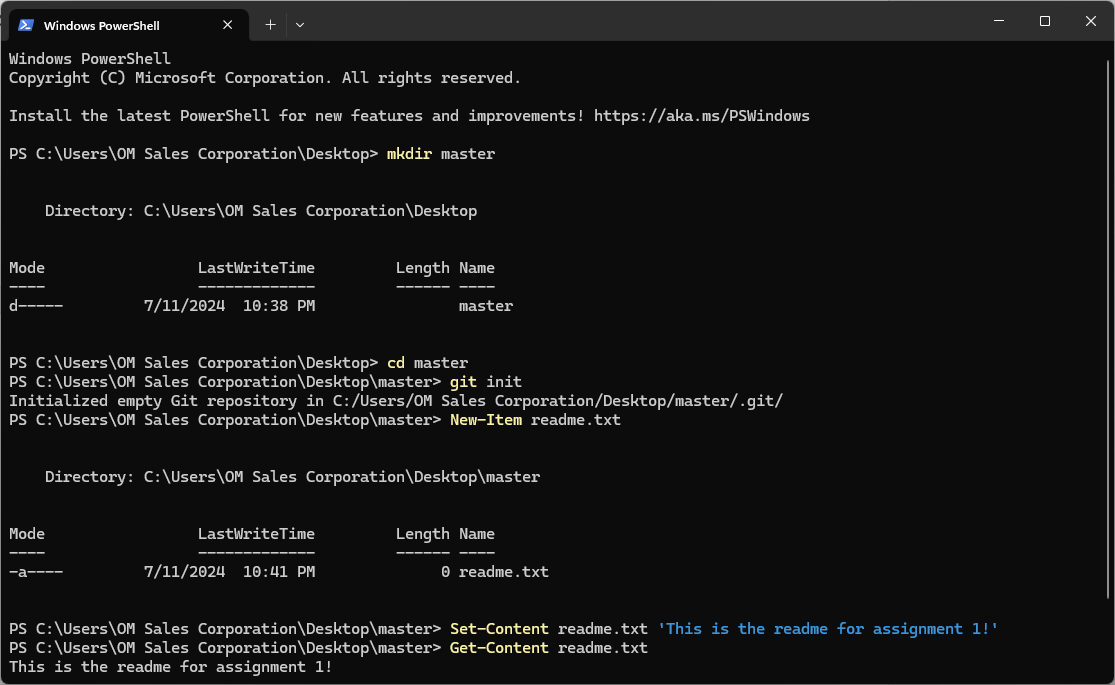
1. Create a Repository
   1. mkdir <repository name>
   2. cd <repository name>
   3. git init
2. Add and Commit Changes
   1. git add .
   2. git commit –m “<message>”
3. Exploring history
   1. git diff <file name>
4. Branching and merging
   1. git checkout –b <branch name>
   2. git checkout <destination branch>
   3. git merge <source branch>
5. Collaborating with Remote Repositories
   1. git remote add <remote name> <repository URL>
   2. git fetch <remote name>
6. Push Changes
   1. git push <remote name> <branch name>

GitHub Operations

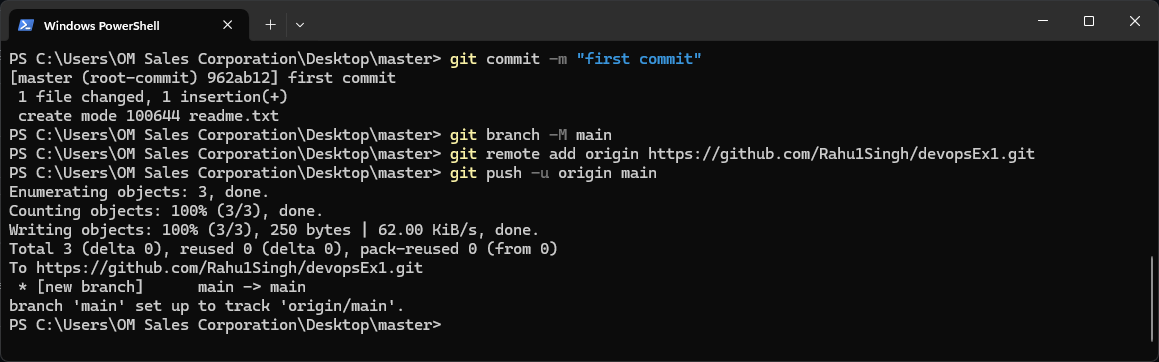
1. Cloning a repository
   1. git clone <repository URL>
2. Making Changes and Creating a Branch
   1. git checkout –b <branch name>
3. Pushing Changes to GitHub
   1. git add .
   2. git commit –m “<message>”
4. Collaborating through Pull Requests
   1. git pull <branch name>
5. Syncing Changes
   1. git checkout <destination branch>
   2. git push

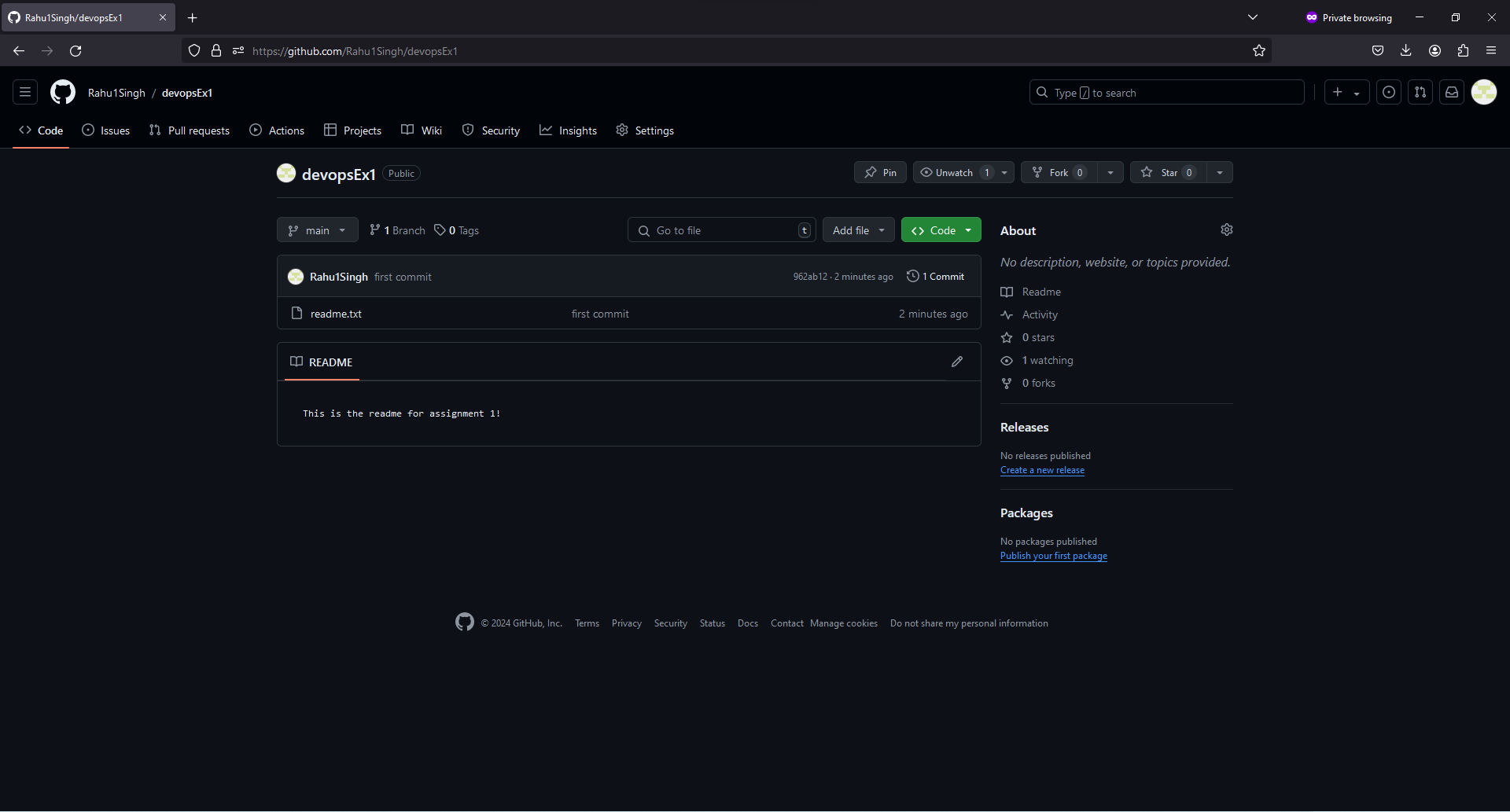
**Output:**

Intitalize git and add content

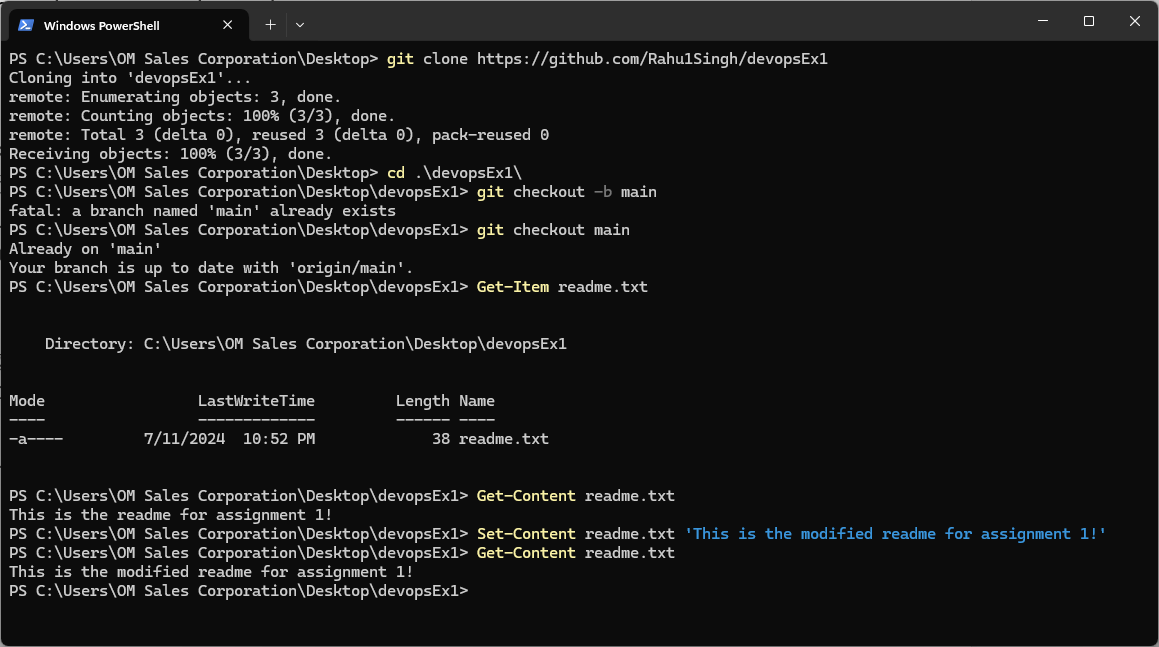
****

Add, commit and push changes into a remote

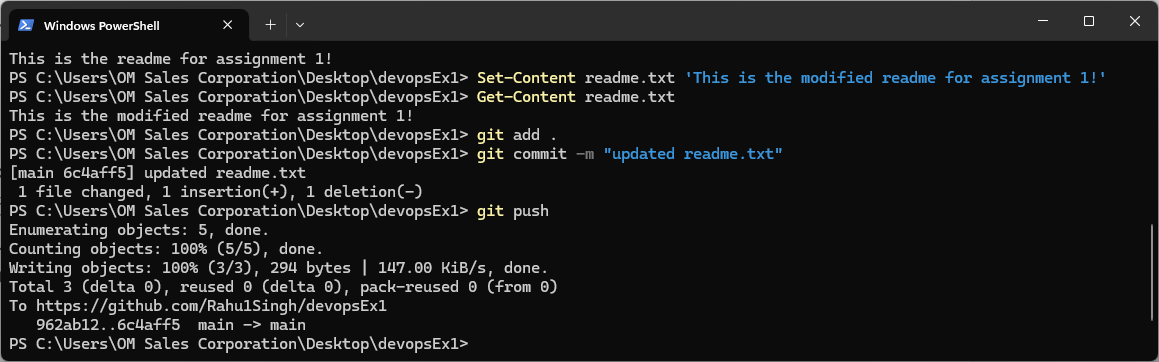
****

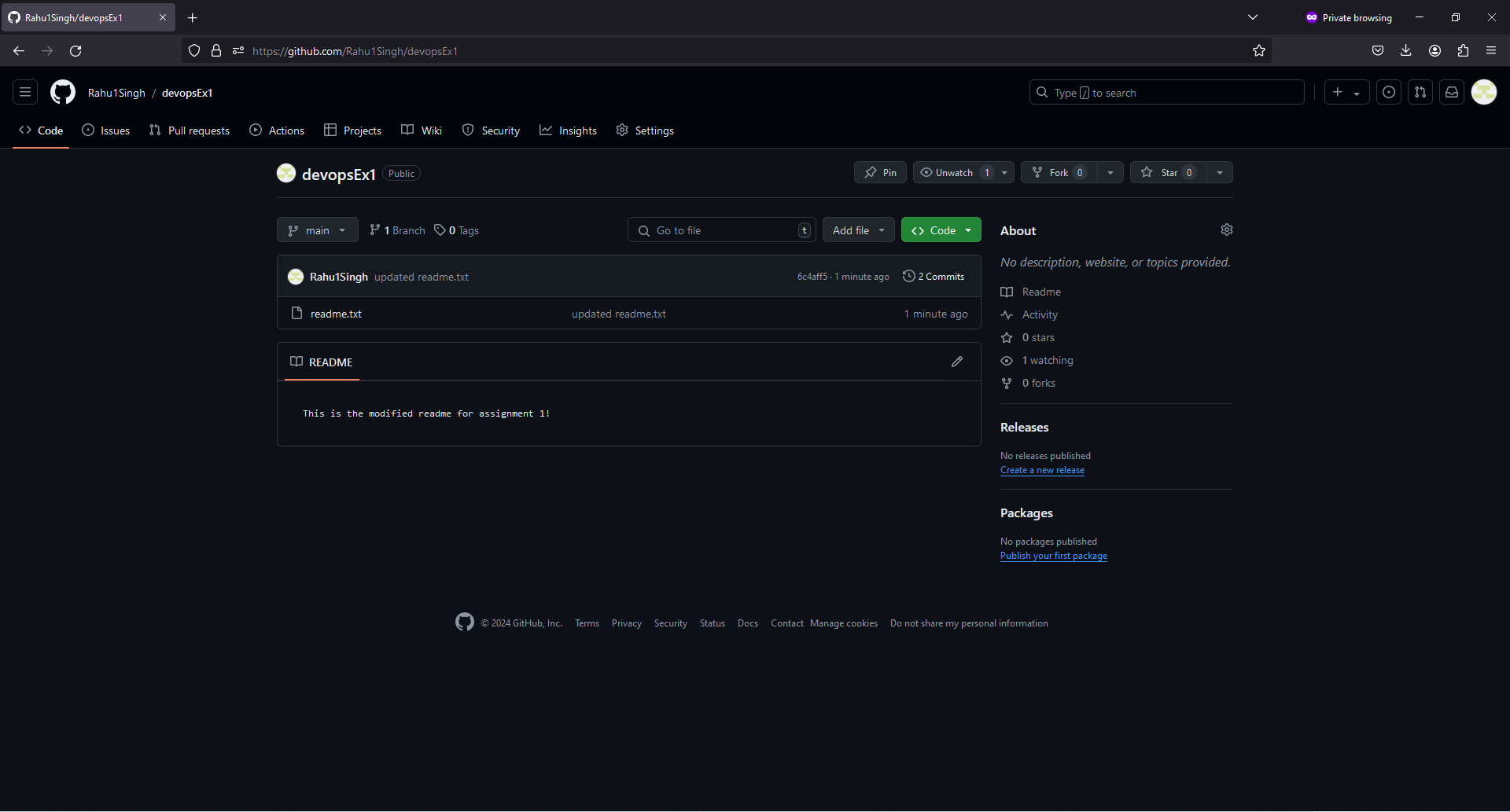
****

Cloning a remote and modifying content

****

Adding, committing and pushing changes into remote

****

****

**Result:**

Basic Git and GitHub operations were learnt and implemented.

|  |  |
| --- | --- |
| **Ex. No. 2** | GitLab Operations |
|  |

**Aim:**

To implement the various GitLab operations using Git.

**Algorithm:**

Git Commands

1. Cloning the repo and editing files
   1. git clone “<repo\_url>”
   2. git status
2. Committing changes and pushing to branch
   1. git branch <branch\_name>
   2. git checkout <branch\_name>
   3. git add <file\_name>
   4. git commit -m “<message>”
   5. git push -u <remote\_name> <branch\_name>
3. Updating branch with changes
   1. git pull <remote\_name> <branch\_name>

**Output:**

Create repository in GitLab

**A screenshot of a computer

Description automatically generated**

Cloning the repository

A screen shot of a computer

Description automatically generated

Creating/Editing Files

**A screenshot of a computer

Description automatically generated**

Committing and pushing changes to repository

**A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer

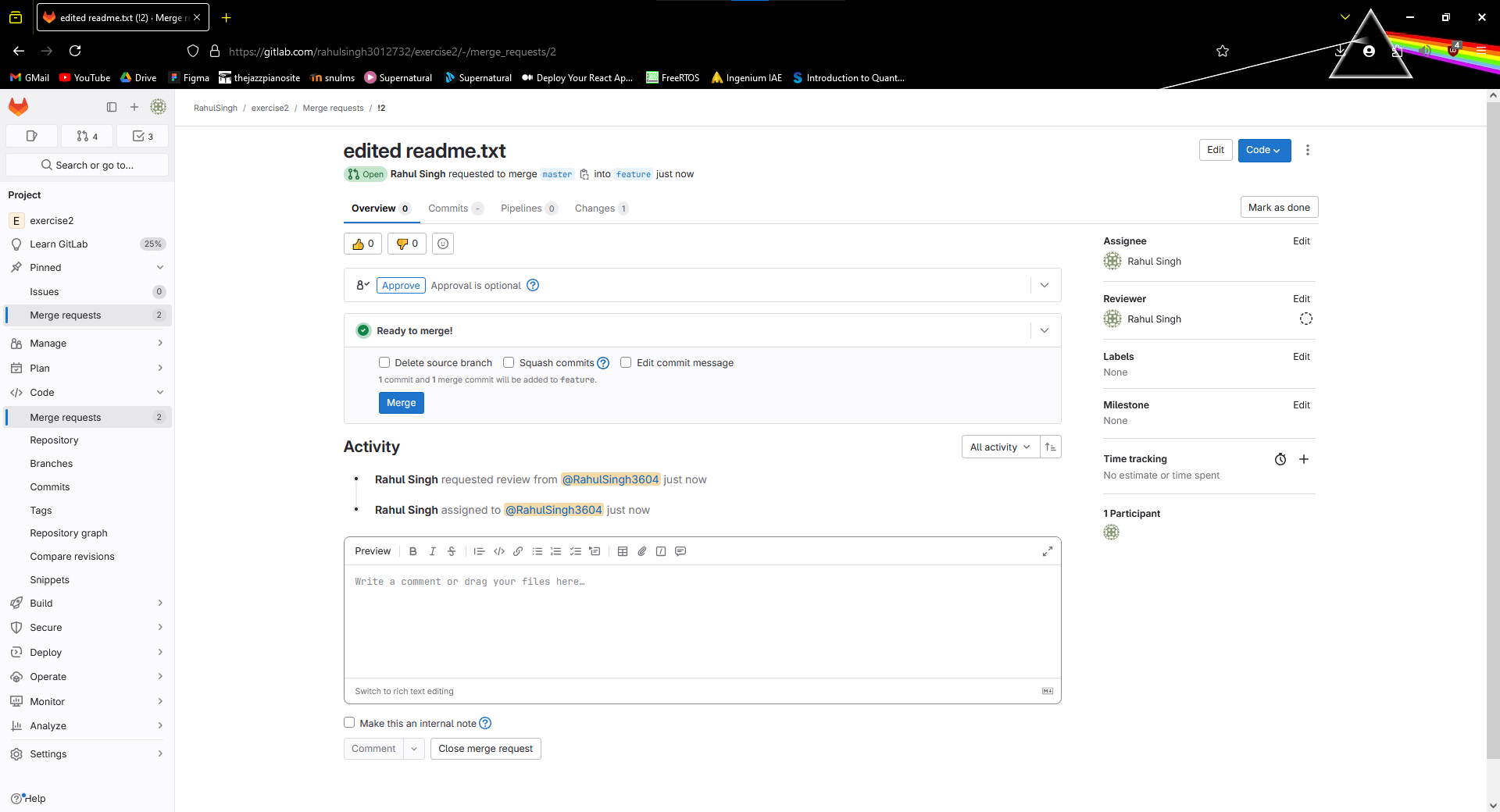
Description automatically generated**

Verifying in GitLab

A screenshot of a computer

Description automatically generated

Creating a merge request

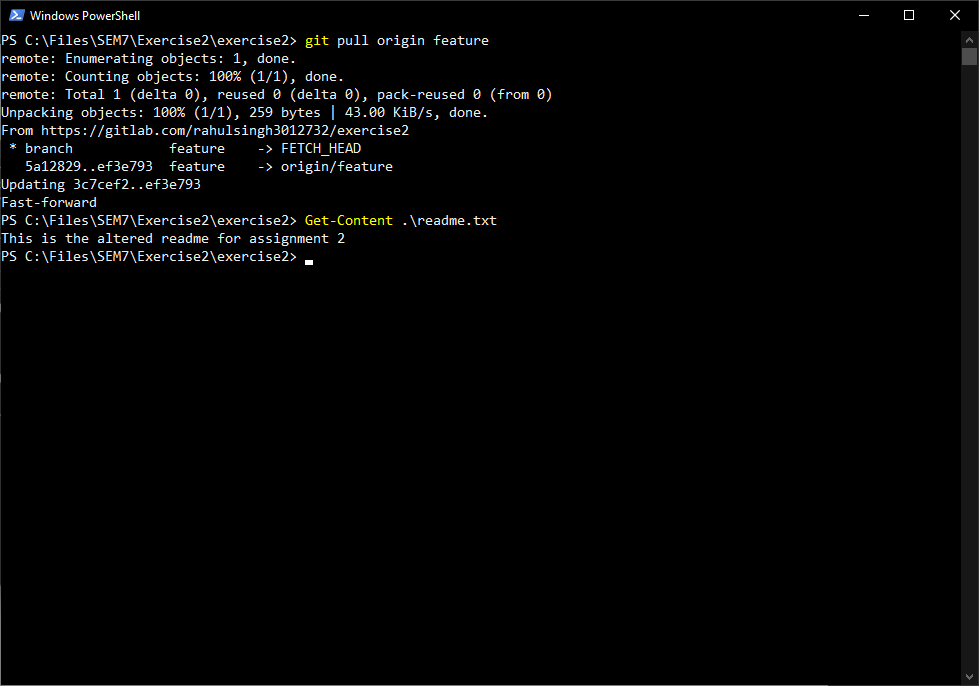
****

Completing the merge

**A screenshot of a computer

Description automatically generated**

Updating the local repository

****

**Result:**

GitLab was explored and git commands were used to implement its operations.

|  |  |
| --- | --- |
| **Ex. No. 3** | BitBucket Operations |
|  |

**Aim:**

To implement the various BitBucket operations using Git.

**Algorithm:**

Git Commands

1. Cloning the repo and editing files
   1. git clone “<repo\_url>”
   2. git status
2. Committing changes and pushing to branch
   1. git branch <branch\_name>
   2. git checkout <branch\_name>
   3. git add <file\_name>
   4. git commit -m “<message>”
   5. git push -u <remote\_name> <branch\_name>
3. Updating branch with changes
   1. git pull <remote\_name> <branch\_name>

**Output:**

Create repository using bitbucket

**A screenshot of a computer

Description automatically generated**

Cloning the repository

**A screen shot of a computer

Description automatically generated**

Creating/Editing Files

****

Committing and pushing changes

**A screen shot of a computer program

Description automatically generated**

Creating a pull request

**A screenshot of a computer

Description automatically generated**

Completing pull and merge

**A screenshot of a computer

Description automatically generated**Updating the local repository

****

**Result:**

BitBucket was explored and git commands were used to implement its operations.

|  |  |
| --- | --- |
| **Ex. No. 4** | **Maven POM** |
|  |

**Aim:**

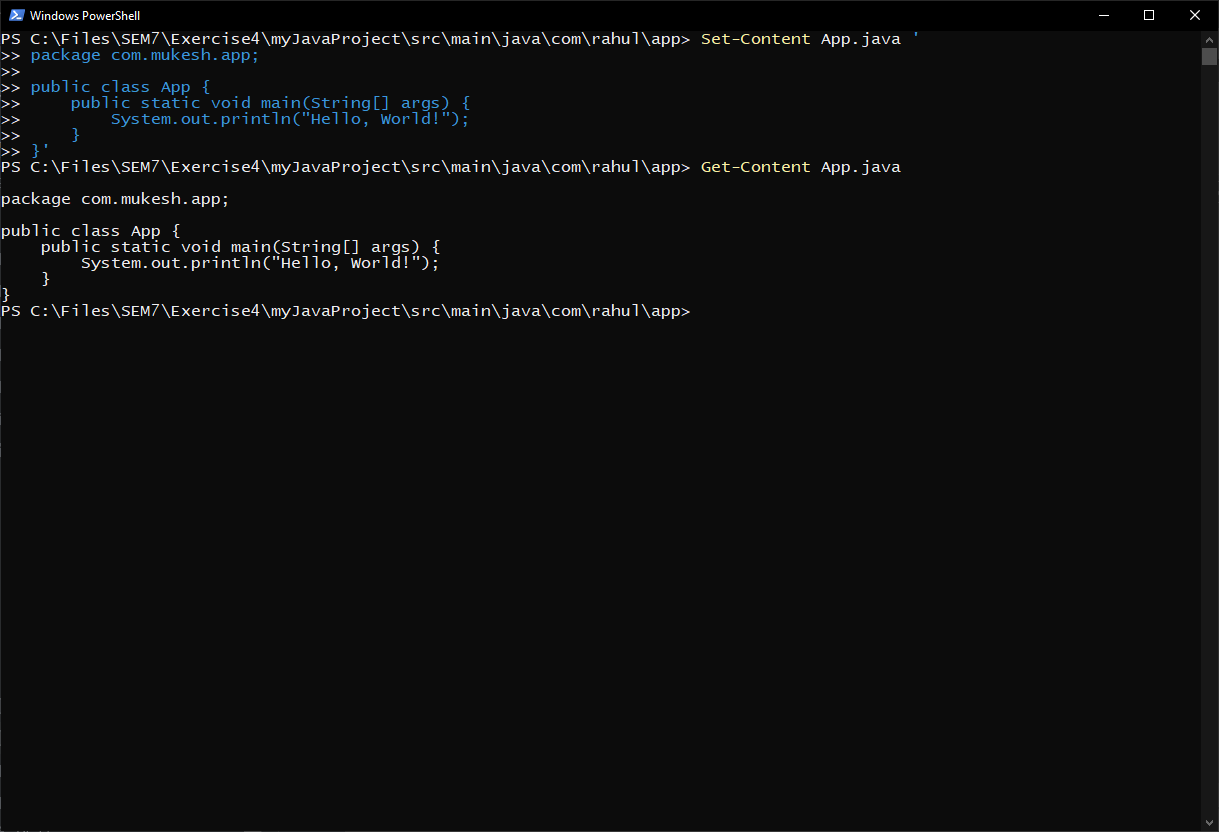
To set up a Java Application with Git and Maven and experiment with Inheritance and Aggregation.

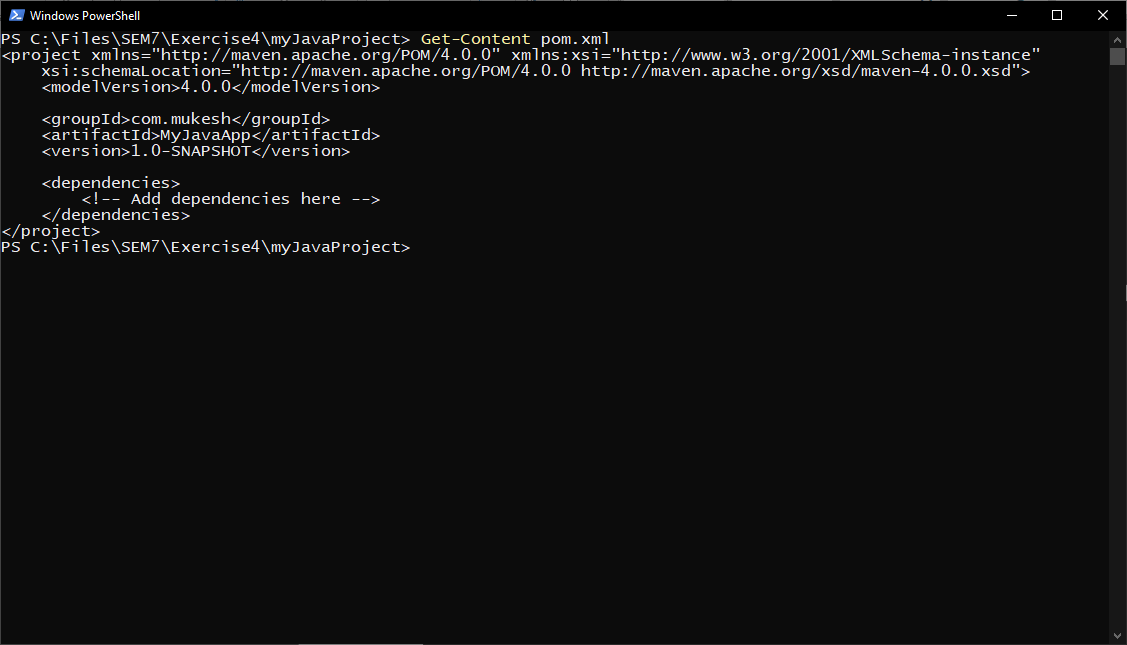
**Algorithm:**

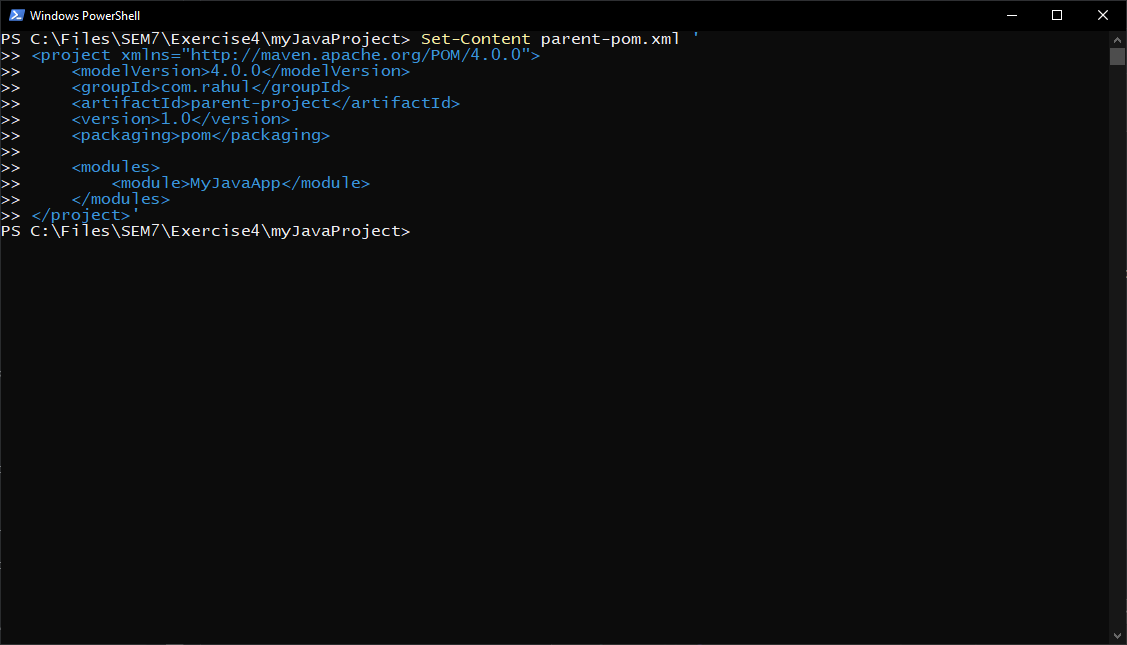
1. Set up a Java Application
   1. Set up a Java project.
   2. Set up the Java application.
2. Working with Maven
   1. Create the pom.xml file.
   2. Perform Inheritance by adding a parent-pom.xml.
   3. Perform Aggregation by adding multiple modules in the pom.xml.

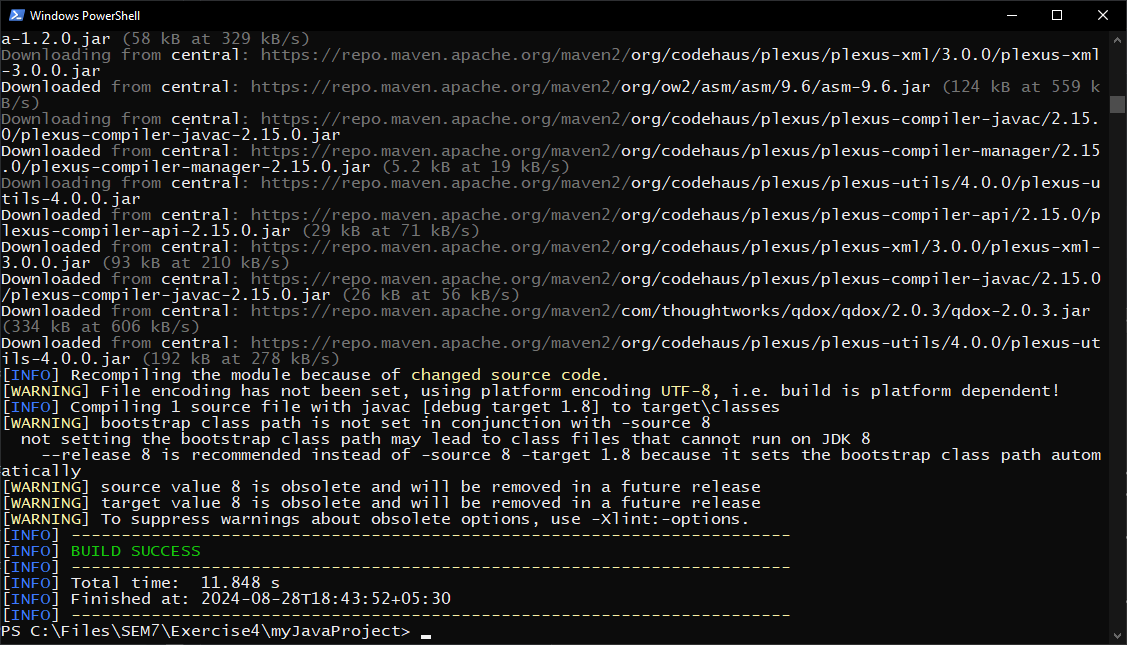
**Output:**











**Results:**

Inheritance and Aggregation were in Maven were experimented with.

|  |  |
| --- | --- |
| **Ex. No. 5** | **Maven POM** |
|  |

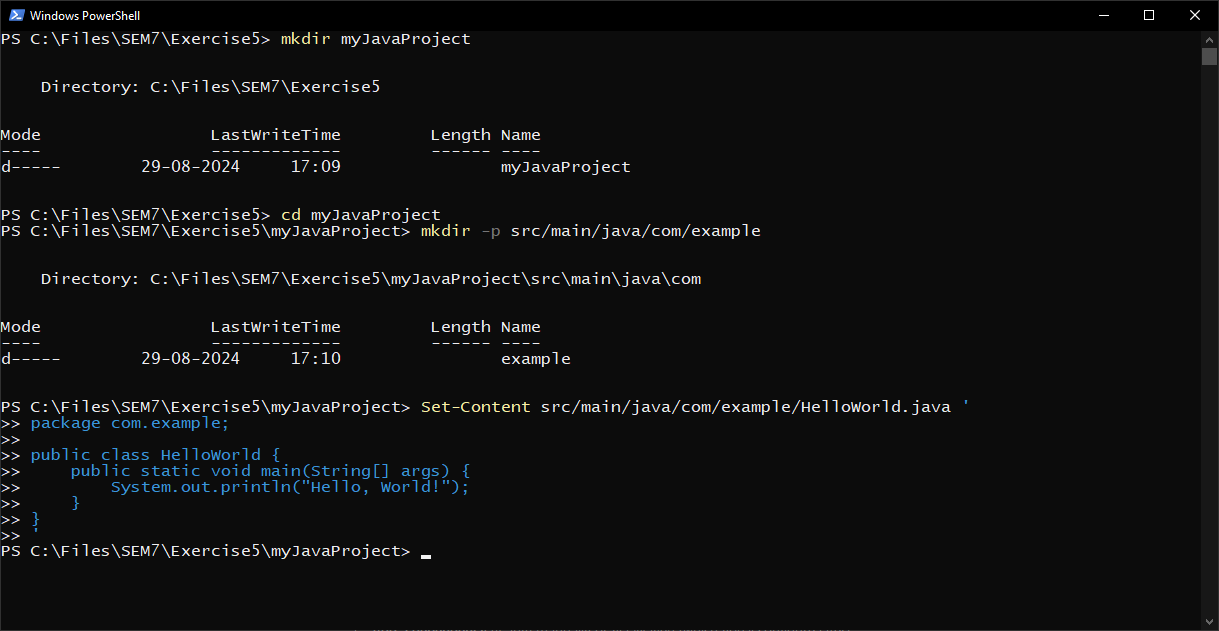
**Aim:**

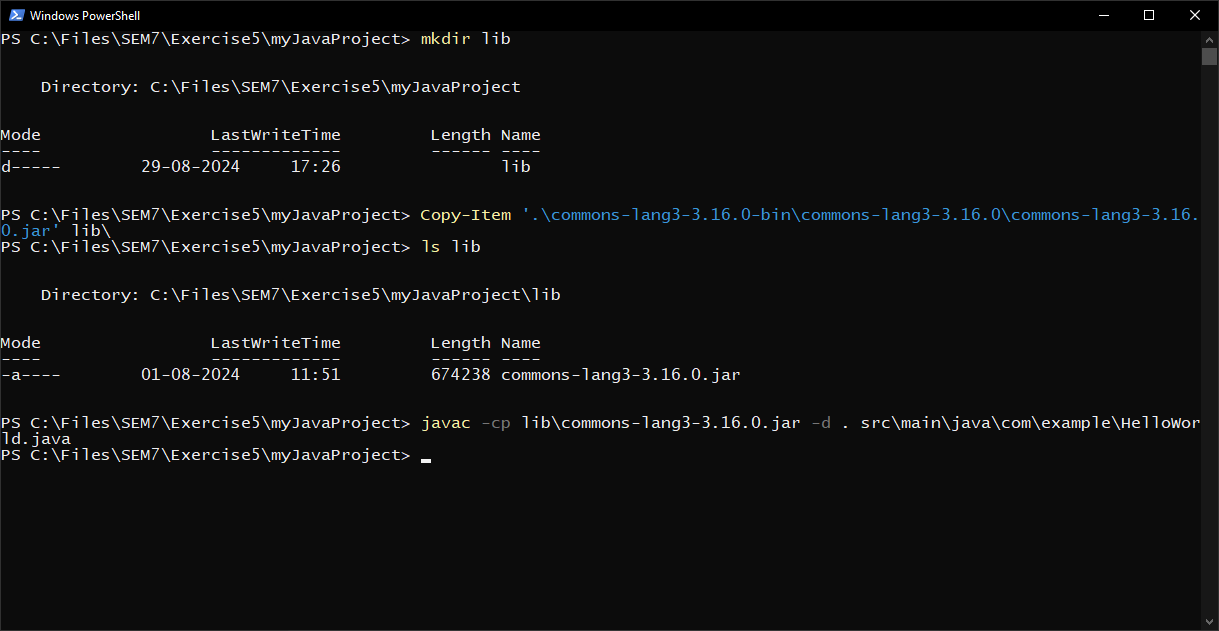
Creating and Managing a Java Project Manually and with Maven

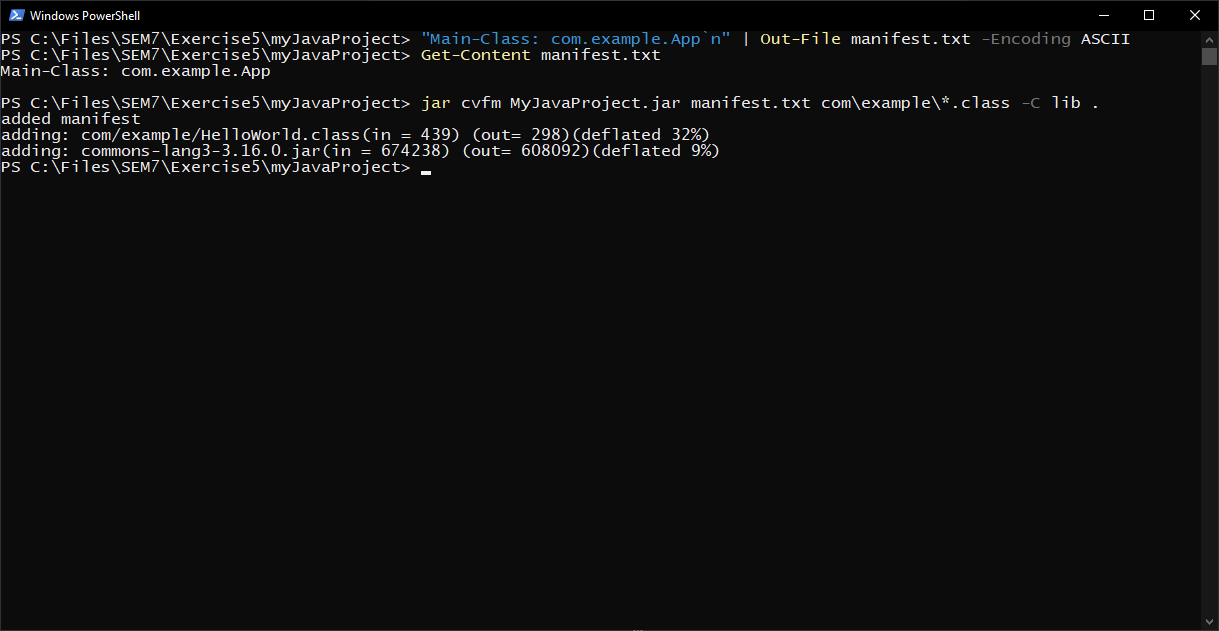
**Algorithm:**

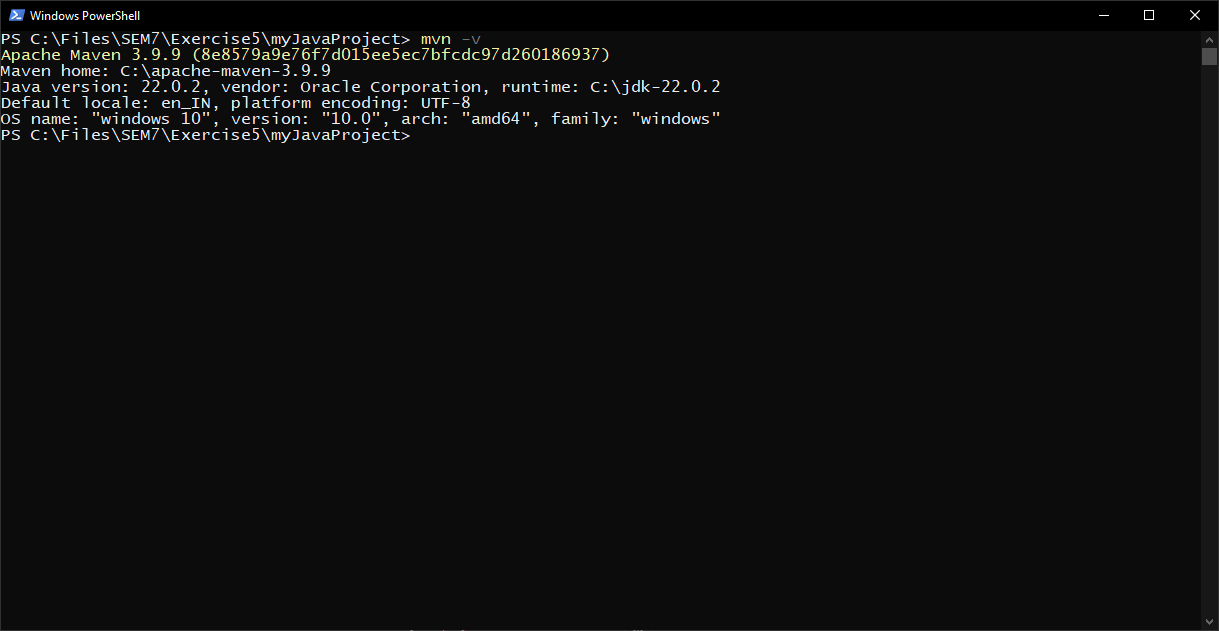
1. Create a java project.
2. Package the project with dependencies.
3. Compile and build the JAR file.
4. Check the maven version.
5. Creating the pom.xml
6. Running the clean lifecycle.
7. Exploring plugins and goals associated with the clean lifecycle.

**Output:**

****

****

****

****

|  |  |
| --- | --- |
| **Ex. No. 6** | **Maven POM Files** |
|  |

**1. Customizing Maven Build Lifecycle**

To execute unit tests only during the integration-test phase:

<build>

    <plugins>

        <plugin>

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

            <artifactId>maven-surefire-plugin</artifactId>

            <version>3.0.0-M5</version>

            <configuration>

                <skip>true</skip>

            </configuration>

            <executions>

                <execution>

                    <id>unit-tests</id>

                    <phase>integration-test</phase>

                    <goals>

                        <goal>test</goal>

                    </goals>

                    <configuration>

                        <skip>false</skip>

                    </configuration>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration skips tests during the default test phase and executes them in the integration-test phase.

**2. Adding External Dependencies and Managing Versions**

To manage Spring Framework dependency across multiple modules:

In the parent pom.xml:

<properties>

    <spring.version>5.3.10</spring.version>

</properties>

<dependencyManagement>

    <dependencies>

        <dependency>

            <groupId>org.springframework</groupId>

            <artifactId>spring-framework-bom</artifactId>

            <version>${spring.version}</version>

            <type>pom</type>

            <scope>import</scope>

        </dependency>

    </dependencies>

</dependencyManagement>

In child module pom.xml files:

<dependencies>

    <dependency>

        <groupId>org.springframework</groupId>

        <artifactId>spring-core</artifactId>

    </dependency>

    <!-- Other Spring modules as needed -->

</dependencies>

**3. Skipping Tests During Build**

To skip tests during a Maven build, use this command:

mvn clean install -DskipTests

Alternatively, you can configure it in the pom.xml:

<properties>

    <maven.test.skip>true</maven.test.skip>

</properties>

**4. Managing Plugins for Code Coverage**

To integrate JaCoCo and enforce 80% code coverage:

<build>

    <plugins>

        <plugin>

            <groupId>org.jacoco</groupId>

            <artifactId>jacoco-maven-plugin</artifactId>

            <version>0.8.7</version>

            <executions>

                <execution>

                    <goals>

                        <goal>prepare-agent</goal>

                    </goals>

                </execution>

                <execution>

                    <id>report</id>

                    <phase>prepare-package</phase>

                    <goals>

                        <goal>report</goal>

                    </goals>

                </execution>

                <execution>

                    <id>check</id>

                    <goals>

                        <goal>check</goal>

                    </goals>

                    <configuration>

                        <rules>

                            <rule>

                                <element>PACKAGE</element>

                                <limits>

                                    <limit>

                                        <counter>LINE</counter>

                                        <value>COVEREDRATIO</value>

                                        <minimum>0.80</minimum>

                                    </limit>

                                </limits>

                            </rule>

                        </rules>

                    </configuration>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

**5. Managing Multi-Module Project Dependencies**

For Module B (the dependency):

<groupId>com.example</groupId>

<artifactId>module-b</artifactId>

<version>1.0-SNAPSHOT</version>

For Module A (dependent on Module B):

<dependencies>

    <dependency>

        <groupId>com.example</groupId>

        <artifactId>module-b</artifactId>

        <version>1.0-SNAPSHOT</version>

    </dependency>

</dependencies>

In the parent pom.xml:

<modules>

    <module>module-b</module>

    <module>module-a</module>

</modules>

This setup ensures Module B is built before Module A, and changes in Module B are reflected in Module A during the build process.

**7. Excluding Transitive Dependencies**

To exclude a specific transitive dependency:

<dependency>

    <groupId>com.example</groupId>

    <artifactId>example-library</artifactId>

    <version>1.0.0</version>

    <exclusions>

        <exclusion>

            <groupId>org.unwanted</groupId>

            <artifactId>unwanted-dependency</artifactId>

        </exclusion>

    </exclusions>

</dependency>

Using Maven Properties to Manage Project Information:

<properties>

    <project.version>1.0.0-SNAPSHOT</project.version>

    <database.url>jdbc:mysql://localhost:3306/mydb</database.url>

    <java.version>11</java.version>

</properties>

<!-- Usage example -->

<version>${project.version}</version>

<configuration>

    <url>${database.url}</url>

</configuration>

**8. Enforcing a Minimum Java Version**

To ensure the project builds with Java 11 or higher:

<build>

    <plugins>

        <plugin>

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

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

            <version>3.8.1</version>

            <configuration>

                <release>11</release>

            </configuration>

        </plugin>

        <plugin>

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

            <artifactId>maven-enforcer-plugin</artifactId>

            <version>3.0.0-M3</version>

            <executions>

                <execution>

                    <id>enforce-java</id>

                    <goals>

                        <goal>enforce</goal>

                    </goals>

                    <configuration>

                        <rules>

                            <requireJavaVersion>

                                <version>[11,)</version>

                            </requireJavaVersion>

                        </rules>

                    </configuration>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

**9. Customizing the Default Build Lifecycle**

To run Checkstyle before the test phase:

<build>

    <plugins>

        <plugin>

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

            <artifactId>maven-checkstyle-plugin</artifactId>

            <version>3.1.2</version>

            <executions>

                <execution>

                    <id>verify-style</id>

                    <phase>process-classes</phase>

                    <goals>

                        <goal>check</goal>

                    </goals>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration binds the Checkstyle check to the process-classes phase, which occurs before the test phase in the default lifecycle.

**10. Skipping Specific Lifecycle Phases**

To skip the test phase but run integration tests:

mvn clean install -DskipTests -Pintegration-test

Assuming you have a profile for integration tests:

<profiles>

    <profile>

        <id>integration-test</id>

        <build>

            <plugins>

                <plugin>

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

                    <artifactId>maven-failsafe-plugin</artifactId>

                    <version>3.0.0-M5</version>

                    <executions>

                        <execution>

                            <goals>

                                <goal>integration-test</goal>

                                <goal>verify</goal>

                            </goals>

                        </execution>

                    </executions>

                </plugin>

            </plugins>

        </build>

    </profile>

</profiles>

**11. Binding Goals to Custom Phases**

To run Flyway before integration tests in specific environments:

<profiles>

    <profile>

        <id>staging</id>

        <activation>

            <property>

                <name>env</name>

                <value>staging</value>

            </property>

        </activation>

        <build>

            <plugins>

                <plugin>

                    <groupId>org.flywaydb</groupId>

                    <artifactId>flyway-maven-plugin</artifactId>

                    <version>7.7.3</version>

                    <executions>

                        <execution>

                            <phase>pre-integration-test</phase>

                            <goals>

                                <goal>migrate</goal>

                            </goals>

                        </execution>

                    </executions>

                </plugin>

            </plugins>

        </build>

    </profile>

</profiles>

Activate this profile with: `mvn clean install -Pstaging -Denv=staging`

**12. Handling Failed Builds**

To troubleshoot and focus on the failing module:

1. Identify the failing module from the build output.

2. Use the following command to build only the failing module and its dependencies:

mvn clean install -pl :failing-module -am

This command uses the `-pl` option to specify the project list (in this case, just the failing module) and the `-am` option to build all modules this module depends on.

By using these options, you can isolate the build to the problematic module and its dependencies, making it easier to identify and fix the issue without rebuilding tHere are the answers to your additional Maven-related questions, with relevant code snippets in markdown format:

**13. Using the Maven Site Lifecycle**

To generate a project site automatically after a successful build:

<build>

    <plugins>

        <plugin>

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

            <artifactId>maven-site-plugin</artifactId>

            <version>3.9.1</version>

            <executions>

                <execution>

                    <id>generate-site</id>

                    <phase>post-integration-test</phase>

                    <goals>

                        <goal>site</goal>

                    </goals>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration binds the site generation to the post-integration-test phase, ensuring it runs after a successful build.

**14. Multi-Module Build with Independent Modules**

Structure your parent pom.xml like this:

<modules>

    <module>module-a</module>

    <module>module-b</module>

    <module>module-c</module>

</modules>

To build specific modules independently:

mvn clean install -pl :module-a,:module-c

This command builds only modules A and C.

**15. Resolving Dependency Conflicts**

To resolve dependency conflicts:

1. Use dependency management:

<dependencyManagement>

    <dependencies>

        <dependency>

            <groupId>com.example</groupId>

            <artifactId>conflicting-library</artifactId>

            <version>2.0.0</version>

        </dependency>

    </dependencies>

</dependencyManagement>

2. Exclude conflicting transitive dependencies:

<dependency>

    <groupId>com.example</groupId>

    <artifactId>example-library</artifactId>

    <version>1.0.0</version>

    <exclusions>

        <exclusion>

            <groupId>com.example</groupId>

            <artifactId>conflicting-library</artifactId>

        </exclusion>

    </exclusions>

</dependency>

3. Use the Maven Enforcer Plugin to fail the build on dependency convergence issues:

<plugin>

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

    <artifactId>maven-enforcer-plugin</artifactId>

    <version>3.0.0-M3</version>

    <executions>

        <execution>

            <id>enforce</id>

            <goals>

                <goal>enforce</goal>

            </goals>

            <configuration>

                <rules>

                    <dependencyConvergence/>

                </rules>

            </configuration>

        </execution>

    </executions>

</plugin>

**16. Ensuring a Clean Build for a Multi-Module Project**

To ensure a clean build across all modules, use this command:

mvn clean install

This command will first execute the clean phase for all modules before starting the build process.

**17. Adding a Custom Clean Goal**

To clean up custom temporary files:

<build>

    <plugins>

        <plugin>

            <artifactId>maven-clean-plugin</artifactId>

            <version>3.1.0</version>

            <configuration>

                <filesets>

                    <fileset>

                        <directory>${project.basedir}/temp</directory>

                        <includes>

                            <include>\*\*/\*</include>

                        </includes>

                    </fileset>

                </filesets>

            </configuration>

        </plugin>

    </plugins>

</build>

**18. Handling Dependencies that Generate Files**

To clean up files generated by a third-party dependency:

<build>

    <plugins>

        <plugin>

            <artifactId>maven-clean-plugin</artifactId>

            <version>3.1.0</version>

            <configuration>

                <filesets>

                    <fileset>

                        <directory>${project.basedir}/generated-files</directory>

                        <includes>

                            <include>\*\*/\*</include>

                        </includes>

                    </fileset>

                </filesets>

            </configuration>

        </plugin>

    </plugins>

</build>

**19. Automatically Cleaning Before Testing**

To ensure clean runs before test automatically:

<build>

    <plugins>

        <plugin>

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

            <artifactId>maven-surefire-plugin</artifactId>

            <version>3.0.0-M5</version>

            <configuration>

                <runOrder>alphabetical</runOrder>

            </configuration>

            <executions>

                <execution>

                    <id>default-test</id>

                    <phase>test</phase>

                    <goals>

                        <goal>test</goal>

                    </goals>

                    <configuration>

                        <runOrder>alphabetical</runOrder>

                    </configuration>

                </execution>

            </executions>

        </plugin>

        <plugin>

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

            <artifactId>maven-clean-plugin</artifactId>

            <version>3.1.0</version>

            <executions>

                <execution>

                    <id>auto-clean</id>

                    <phase>initialize</phase>

                    <goals>

                        <goal>clean</goal>

                    </goals>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration binds the clean goal to the initialize phase, which runs before the test phase. Now, running `mvn test` will automatically clean before testing.he entire project.

**20. Customizing the Clean Phase with Additional Goals**

To extend the clean lifecycle with additional cleanup actions:

<build>

    <plugins>

        <plugin>

            <artifactId>maven-clean-plugin</artifactId>

            <version>3.1.0</version>

            <configuration>

                <filesets>

                    <fileset>

                        <directory>${user.home}/.cache/myapp</directory>

                        <includes>

                            <include>\*\*/\*</include>

                        </includes>

                    </fileset>

                    <fileset>

                        <directory>${user.home}/logs/myapp</directory>

                        <includes>

                            <include>\*.log</include>

                        </includes>

                    </fileset>

                </filesets>

            </configuration>

        </plugin>

    </plugins>

</build>

**21. Cleaning Dependent Projects in a Multi-Module Build**

Maven automatically handles cleaning of dependent modules. When you run `mvn clean` on a parent project, it will clean all child modules. To ensure this behavior:

1. Ensure your parent POM correctly defines modules:

<modules>

    <module>module-a</module>

    <module>module-b</module>

    <module>module-c</module>

</modules>

2. Define dependencies between modules in their respective POMs:

<dependencies>

    <dependency>

        <groupId>${project.groupId}</groupId>

        <artifactId>module-a</artifactId>

        <version>${project.version}</version>

    </dependency>

</dependencies>

**22. Managing Build Artifacts During the Clean Phase**

To exclude certain directories from the clean phase:

<build>

    <plugins>

        <plugin>

            <artifactId>maven-clean-plugin</artifactId>

            <version>3.1.0</version>

            <configuration>

                <excludeDefaultDirectories>true</excludeDefaultDirectories>

                <filesets>

                    <fileset>

                        <directory>${project.build.directory}</directory>

                        <excludes>

                            <exclude>downloaded-files/\*\*</exclude>

                        </excludes>

                    </fileset>

                </filesets>

            </configuration>

        </plugin>

    </plugins>

</build>

**23. Reusing Common Build Configurations**

Structure a parent POM for shared configurations:

<project>

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>

    <artifactId>parent-project</artifactId>

    <version>1.0.0</version>

    <packaging>pom</packaging>

    <properties>

        <java.version>11</java.version>

        <maven.compiler.source>${java.version}</maven.compiler.source>

        <maven.compiler.target>${java.version}</maven.compiler.target>

    </properties>

    <dependencyManagement>

        <dependencies>

            <!-- Common dependencies -->

        </dependencies>

    </dependencyManagement>

    <build>

        <pluginManagement>

            <plugins>

                <!-- Common plugin configurations -->

            </plugins>

        </pluginManagement>

    </build>

</project>

**24. Overriding Parent Configurations in Child Projects**

To override Java version and add a new plugin in a child POM:

<project>

    <parent>

        <groupId>com.example</groupId>

        <artifactId>parent-project</artifactId>

        <version>1.0.0</version>

    </parent>

    <artifactId>child-project</artifactId>

    <properties>

        <java.version>11</java.version>

    </properties>

    <build>

        <plugins>

            <plugin>

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

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

                <version>3.8.1</version>

                <configuration>

                    <source>${java.version}</source>

                    <target>${java.version}</target>

                </configuration>

            </plugin>

            <plugin>

                <!-- New plugin configuration -->

            </plugin>

        </plugins>

    </build>

</project>

**25. Managing Dependencies in Parent and Child Projects**

To override a dependency version in a child project:

<project>

    <parent>

        <groupId>com.example</groupId>

        <artifactId>parent-project</artifactId>

        <version>1.0.0</version>

    </parent>

    <artifactId>child-project</artifactId>

    <dependencies>

        <dependency>

            <groupId>com.example</groupId>

            <artifactId>overridden-dependency</artifactId>

            <version>2.0.0</version> <!-- Overridden version -->

        </dependency>

    </dependencies>

</project>

This configuration in the child POM will use version 2.0.0 of the dependency, overriding the version specified in the parent POM.

**26. Using Profiles in Parent POM for Environment-Specific Configurations**

In the parent POM:

<profiles>

    <profile>

        <id>dev</id>

        <activation>

            <activeByDefault>true</activeByDefault>

        </activation>

        <properties>

            <db.url>jdbc:mysql://localhost:3306/devdb</db.url>

        </properties>

    </profile>

    <profile>

        <id>prod</id>

        <properties>

            <db.url>jdbc:mysql://prodserver:3306/proddb</db.url>

        </properties>

    </profile>

</profiles>

Child projects can inherit and use these profiles automatically. To activate a specific profile:

mvn clean install -Pprod

**27. Inheriting and Extending Build Plugins**

To extend an inherited plugin configuration in a child POM:

<build>

    <plugins>

        <plugin>

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

            <artifactId>maven-surefire-plugin</artifactId>

            <configuration combine.children="append">

                <includes>

                    <include>\*\*/\*IntegrationTest.java</include>

                </includes>

            </configuration>

            <executions>

                <execution>

                    <id>integration-tests</id>

                    <goals>

                        <goal>test</goal>

                    </goals>

                    <phase>integration-test</phase>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

**28. Managing Parent-Child Relationships in a Multi-Module Project**

In the child POM that requires a new plugin:

<build>

    <plugins>

        <plugin>

            <groupId>org.example</groupId>

            <artifactId>custom-plugin</artifactId>

            <version>1.0.0</version>

            <executions>

                <execution>

                    <goals>

                        <goal>custom-goal</goal>

                    </goals>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration adds the new plugin only to this specific child module.

**29. Inheriting and Customizing Dependency Versions in Child Projects**

In the child POM, to override a dependency version:

<dependencies>

    <dependency>

        <groupId>org.example</groupId>

        <artifactId>example-dependency</artifactId>

        <version>2.0.0</version> <!-- Overridden version -->

    </dependency>

</dependencies>

This will use version 2.0.0 of the dependency, overriding the version specified in the parent POM's `<dependencyManagement>` section.

**30. Customizing the Clean Lifecycle**

In the child POM that requires custom clean actions:

<build>

    <plugins>

        <plugin>

            <artifactId>maven-clean-plugin</artifactId>

            <version>3.1.0</version>

            <executions>

                <execution>

                    <id>custom-clean</id>

                    <phase>clean</phase>

                    <goals>

                        <goal>clean</goal>

                    </goals>

                    <configuration>

                        <filesets>

                            <fileset>

                                <directory>${project.basedir}/custom-dir</directory>

                                <includes>

                                    <include>\*\*/\*</include>

                                </includes>

                            </fileset>

                        </filesets>

                    </configuration>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration adds custom clean actions only for this specific module.

**31. Using Plugin Inheritance with Custom Execution Goals**

In the child POM, to add a custom execution goal for an inherited plugin:

<build>

    <plugins>

        <plugin>

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

            <artifactId>maven-surefire-plugin</artifactId>

            <executions>

                <execution>

                    <id>integration-tests</id>

                    <goals>

                        <goal>test</goal>

                    </goals>

                    <phase>integration-test</phase>

                    <configuration>

                        <includes>

                            <include>\*\*/\*IT.java</include>

                        </includes>

                    </configuration>

                </execution>

            </executions>

        </plugin>

    </plugins>

</build>

This configuration adds an additional execution goal for running integration tests, specific to this child module, while still inheriting the basic plugin configuration from the parent POM.

|  |  |
| --- | --- |
| **Ex. No. 8** | **Jenkins** |
|  |

**Aim:**

To set up a CI/CD pipeline for a web development project using Jenkins, Git, webhooks, and a Python server.

**Algorithm:**

1. **Code Changes:** Developers make changes to the web application source code locally.
2. **Git Repository:** Developers push their changes to a Git repository (GitHub or another service).
3. **Webhook:** A webhook in the Git repository notifies Jenkins whenever changes are pushed.
4. **Jenkins Job:** Jenkins listens for webhook triggers and initiates the CI/CD pipeline when changes are detected.
5. **Build and Test:** Jenkins runs predefined steps, which can include building the application, running tests, and creating artifacts.
6. **Deployment:** If the previous steps succeed, Jenkins deploys the updated application to a Python server.
7. **Verification:** The application is tested locally to ensure it works as expected.

**Procedure:**

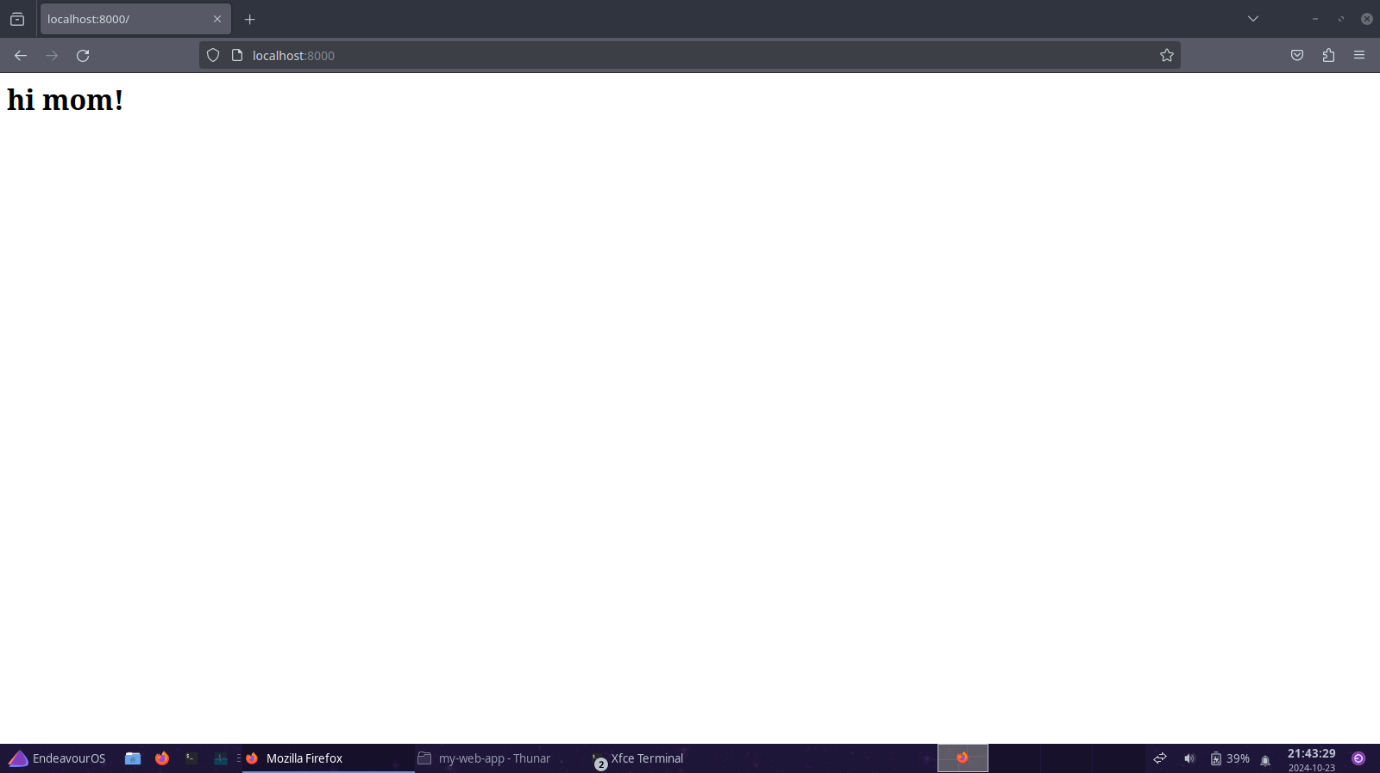
**1. Set Up the Web Application and Python HTTP Server**

- Make sure you have Python installed on your EndeavourOS VM.

- Start a simple Python server for testing:

   python3 -m http.server 8000

This will serve the content of your directory at `http://localhost:8000`.



**2. Set Up a Git Repository**

- Initialize a Git repository in your project directory:

   git init

   git add .

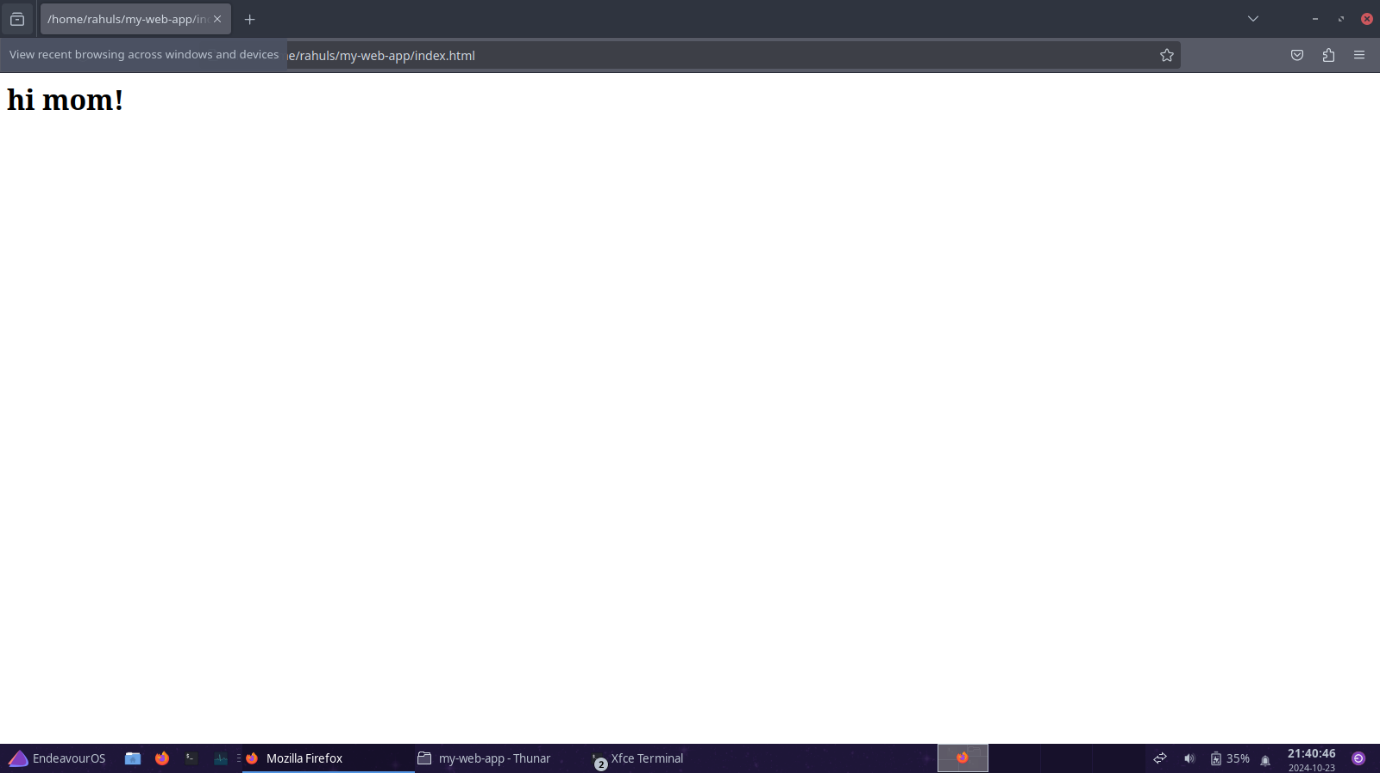
   git commit -m "Initial commit"

- Add your remote repository (if you're using GitHub or another service):

   git branch -M main

   git remote add origin https://github.com/YourUsername/Repository.git

   git push -u origin main



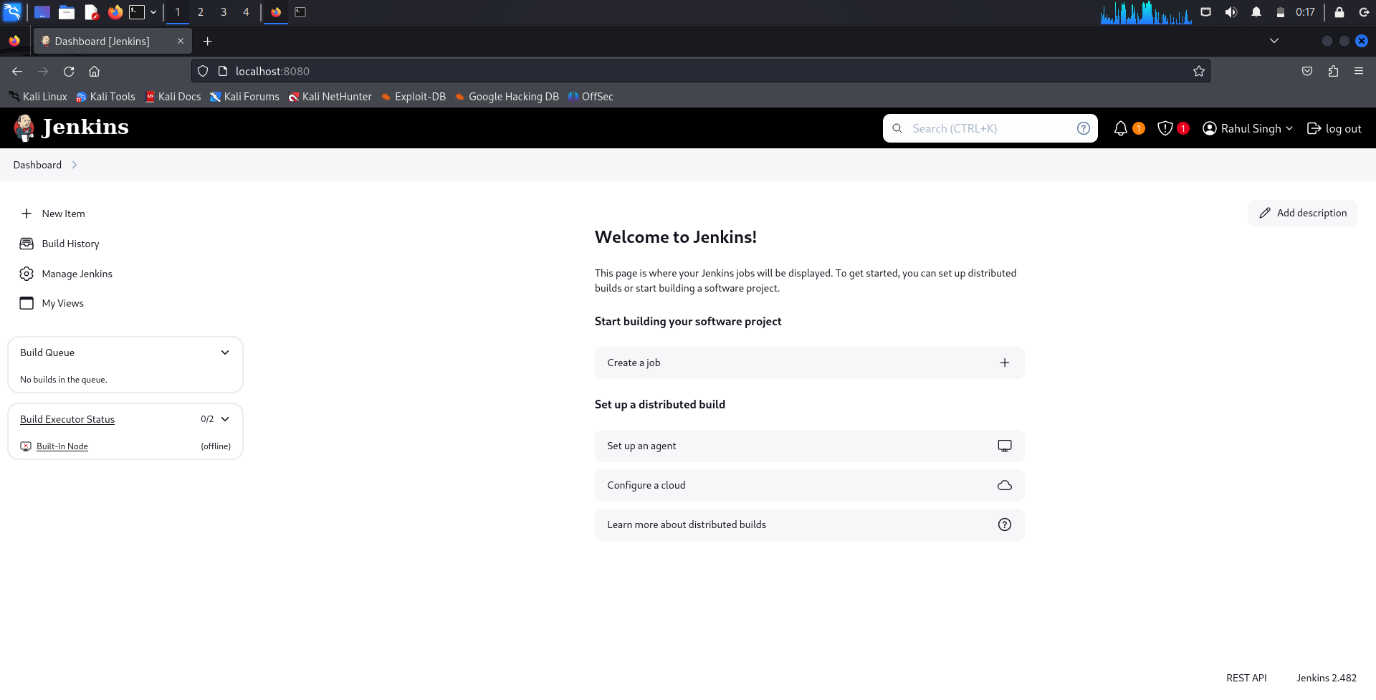
**3. Install and Configure Jenkins**

- Install Jenkins on your EndeavourOS VM (follow the instructions for your distribution).

- Start Jenkins:

   sudo systemctl start Jenkins

- Access Jenkins through your browser at `http://localhost:8080` and complete the initial setup.



**4. Create a Jenkins Job (Freestyle Project)**

1. **Create a new job**:

   - In Jenkins, click "New Item" and select "Freestyle Project."

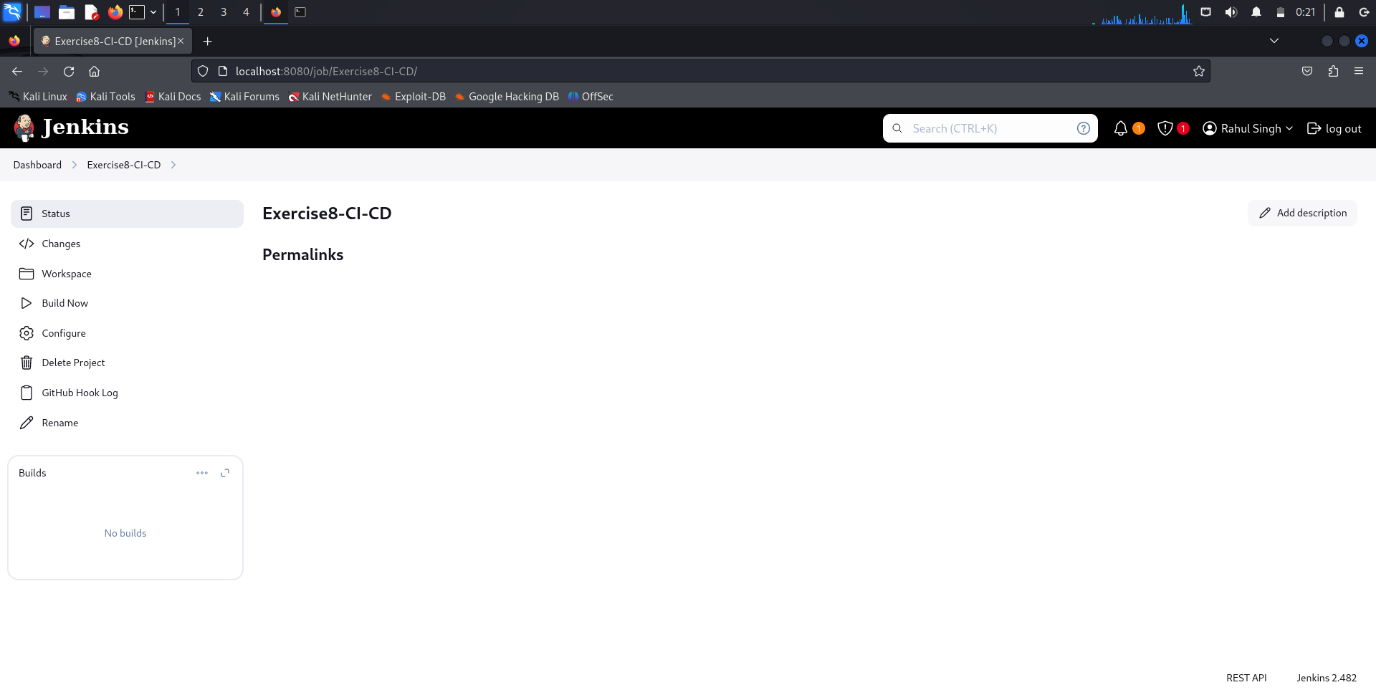
   - Give it a name, then click "OK."

2. **Configure the job**:

   - Under the "Source Code Management" section, choose **Git** and provide your repository URL.

3. **Set up a webhook trigger**:

   - Under "Build Triggers," check the box for **GitHub hook trigger for GITScm polling**.



**5. Set Up the Jenkins Job (Using Execute Shell)**

1. In the job configuration, go to the **\*\*Build\*\*** section.

2. Add a build step of type "Execute shell."

3. Define the build and deployment steps using shell commands. For example:

   # Checkout code from Git

   git pull origin main

   # Deploy the application to the Python server directory

   rm -rf /path/to/your/python/server/directory/\*

   cp -r \* /path/to/your/python/server/directory/

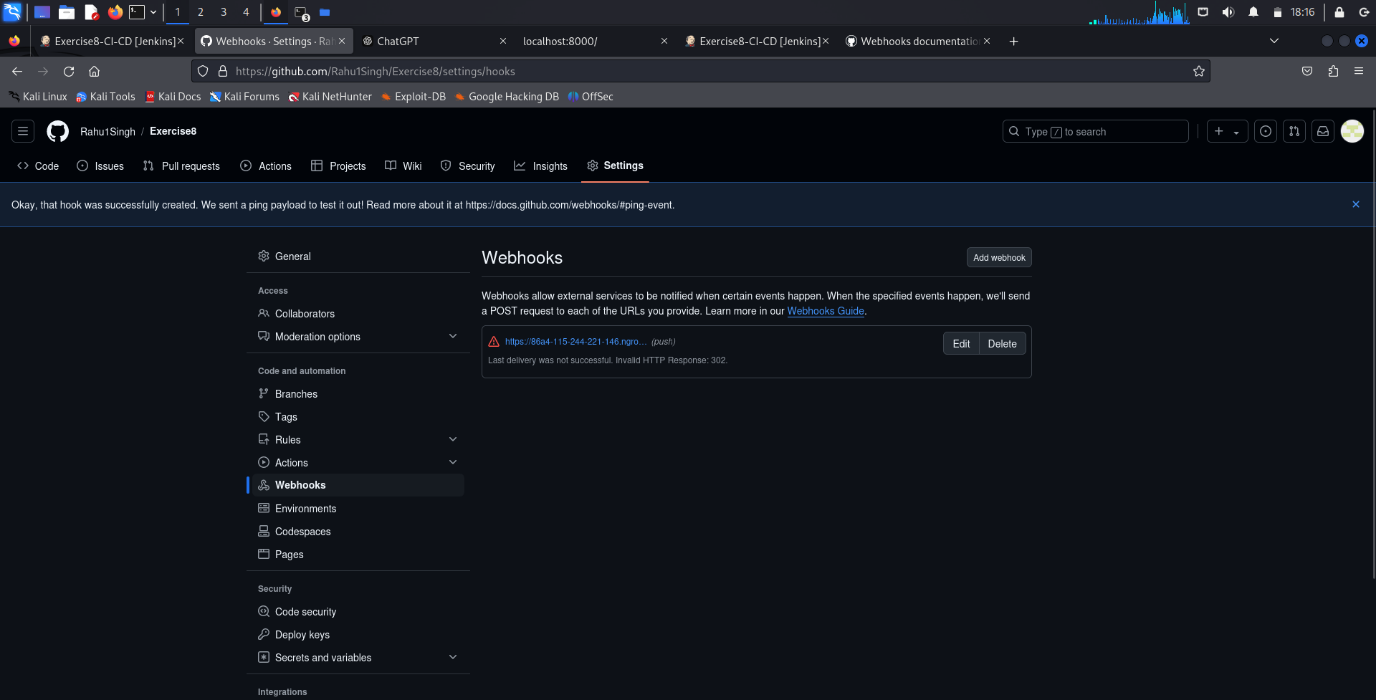
   Replace `/path/to/your/python/server/directory/` with the actual directory being served by your Python server.

**6. Set Up a Webhook in the Git Repository**

1. In your GitHub repository, go to **Settings** > **Webhooks**.

2. Create a new webhook and set the **Payload URL** to `http://your-jenkins-server/github-webhook/`.

3. Set the **Content type** to `application/json` and leave the other settings as default.



**7. Trigger the CI/CD Pipeline**

1. Push changes to your Git repository:

   git add .

   git commit -m "Updated the web app"

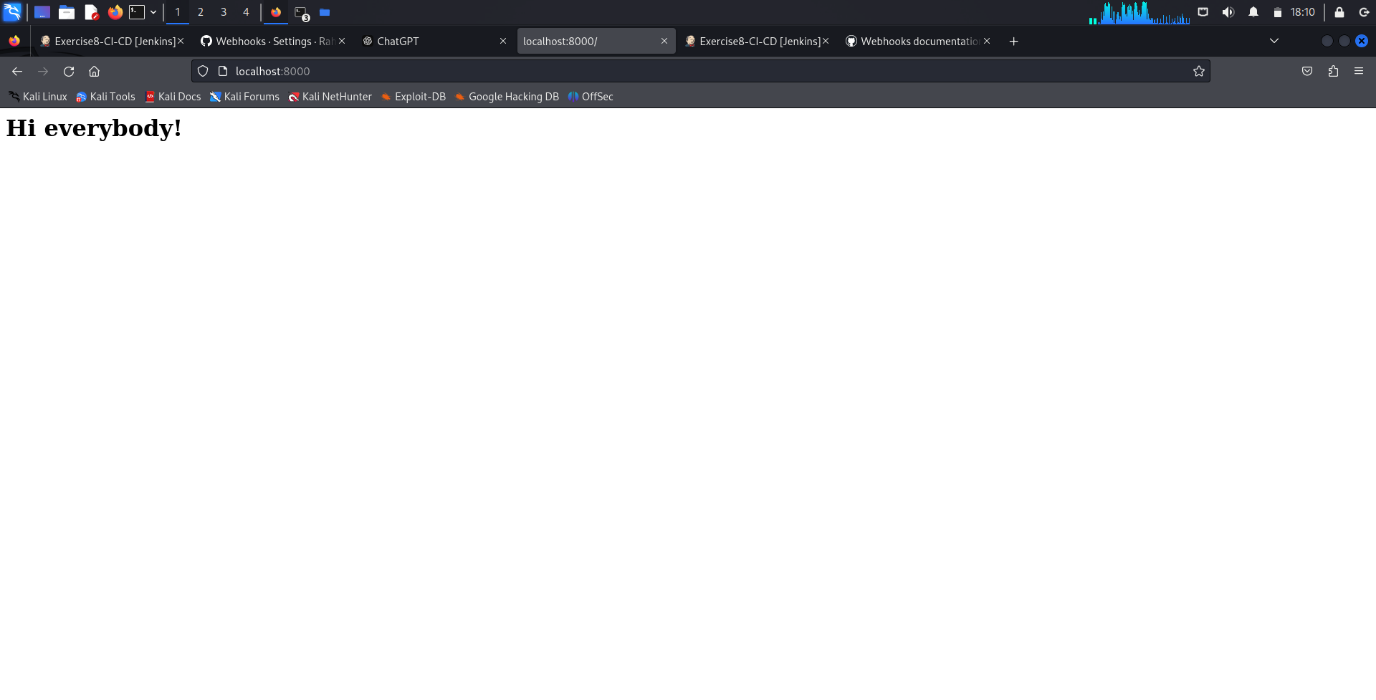
   git push

2. The webhook will automatically trigger the Jenkins job, executing the build and deployment steps.

3. Monitor the job's progress in the Jenkins web interface.

**8. Verify the CI/CD Pipeline**

After Jenkins completes the build and deploy process, visit `http://localhost:8000` to check if the latest version of your web application is deployed on the Python server.



**Result:**

A CI/CD pipeline was implemented using Jenkins.

|  |  |
| --- | --- |
| **Ex. No. 9** | **Docker** |
|  |

**Aim:**

To run an Apache server on a docker container.

**Algorithm:**

**Step 1: Install Docker**

- Visit [Docker's official installation page](https://docs.docker.com/get-docker/) and follow the installation instructions for your operating system.

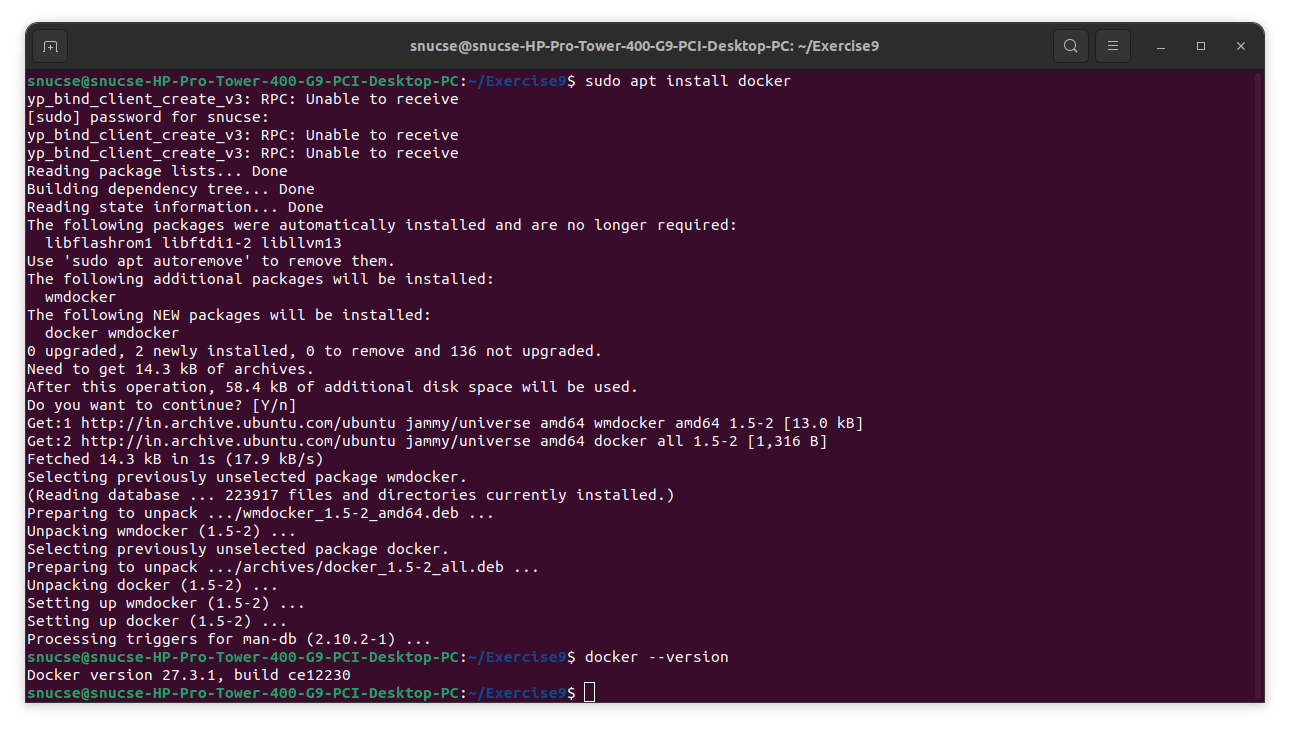
  - **Windows/macOS:** You can download and install Docker Desktop.

  - **Linux:** Install Docker using your distribution's package manager (e.g., `apt` for Ubuntu).

  After installation, you can verify that Docker is running by typing the following command in the terminal or command prompt:

  docker --version

  You should see the Docker version printed.



**Step 2: Create a Simple HTML Page**

1. **Create a directory** on your computer where your project will reside:

   mkdir my-docker-webserver

   cd my-docker-webserver

2. **Create the HTML file**:

   - In the directory, create a file named `index.html` with the content that will be served by the web server.

   - You can use any text editor (VSCode, Notepad, etc.) to create the file. Here's an example of what the `index.html` might look like:

     <!DOCTYPE html>

     <html>

     <head>

         <title>Hi mom</title>

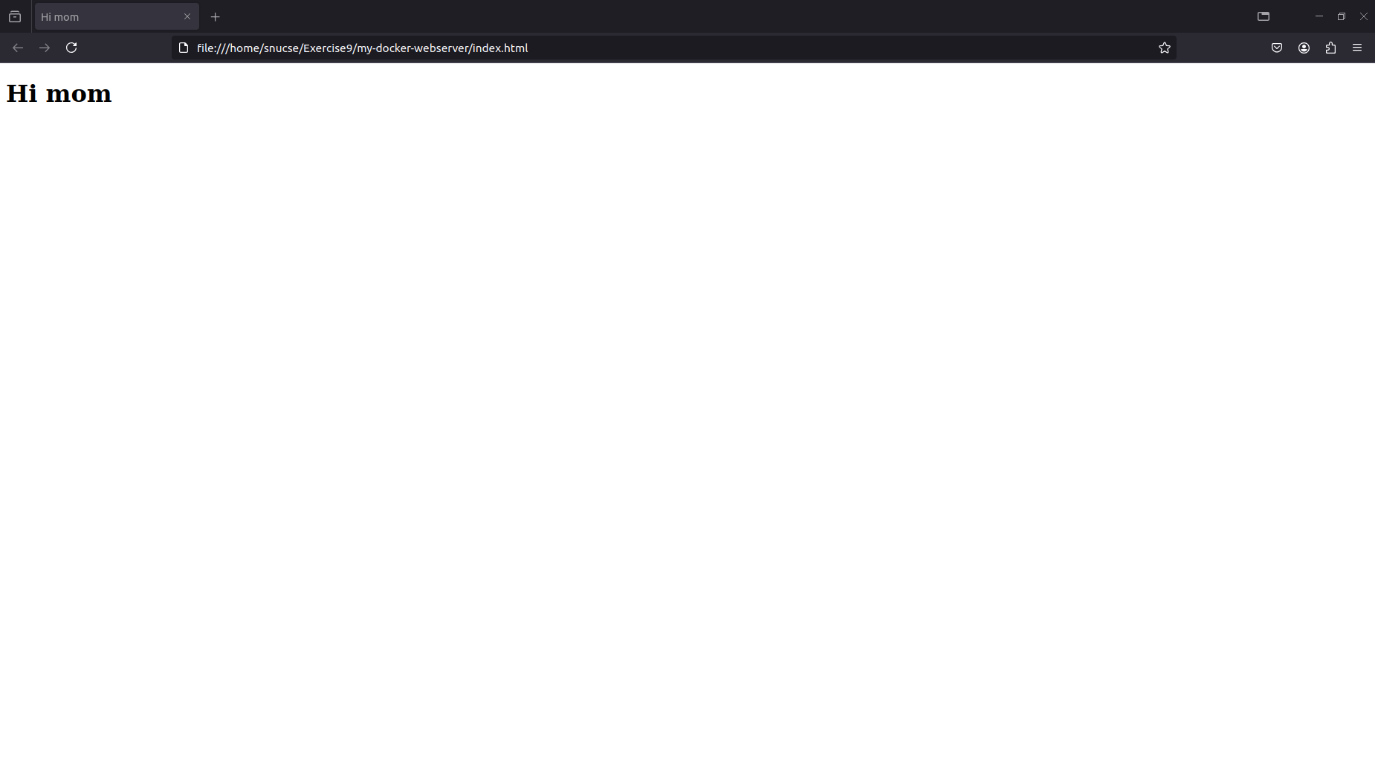
     </head>

     <body>

         <h1>Hi mom</h1>

     </body>

     </html>



**Step 3: Create a Dockerfile**

1. Inside the same `my-docker-webserver` directory, **create a file named `Dockerfile`** (no extension).

2. Open the `Dockerfile` in a text editor and add the following content:

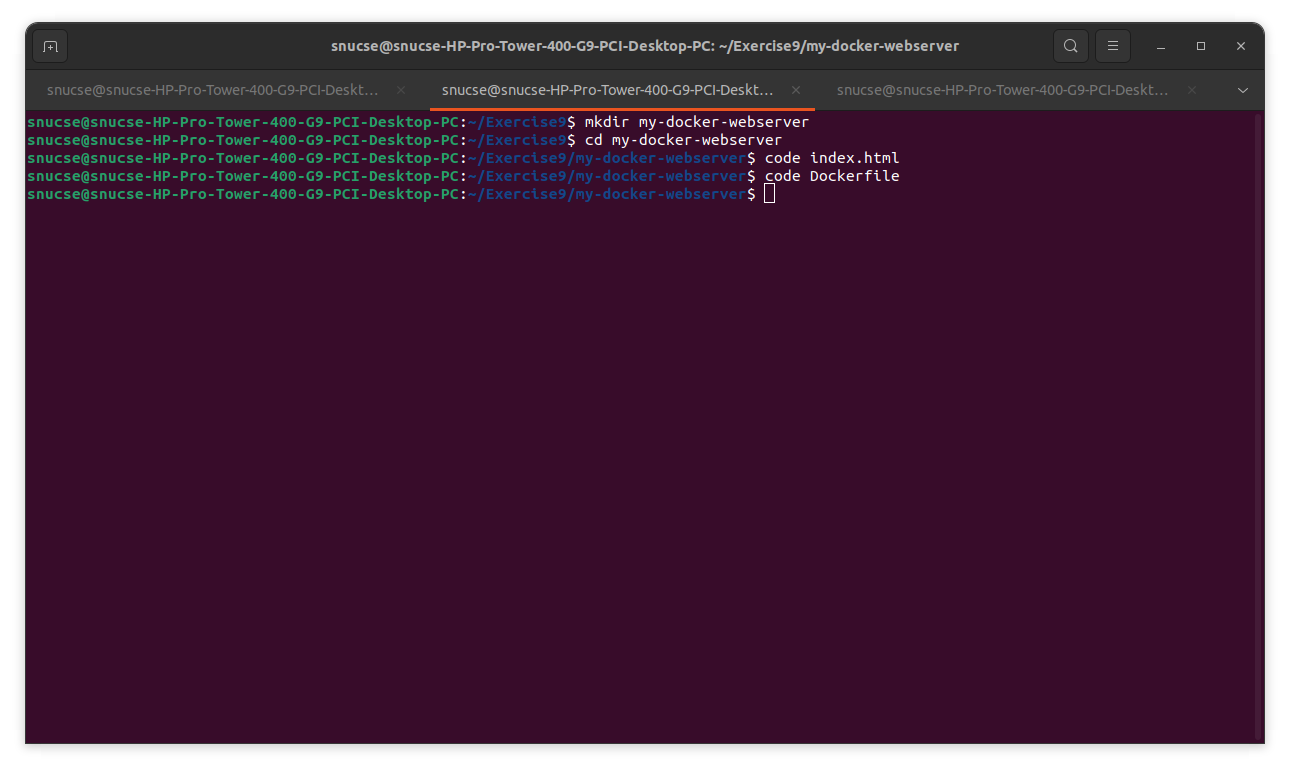
   FROM httpd:2.4

   COPY index.html /usr/local/apache2/htdocs/

This tells Docker to:

- Use the official Apache (`httpd`) image version 2.4 as the base image.

- Copy your `index.html` file into the Apache web server's document root directory (`/usr/local/apache2/htdocs/`).



**Step 4: Build the Docker Image**

Now that you have your `Dockerfile` and `index.html`, it’s time to build the Docker image.

1. In the terminal, make sure you are still in the `my-docker-webserver` directory.

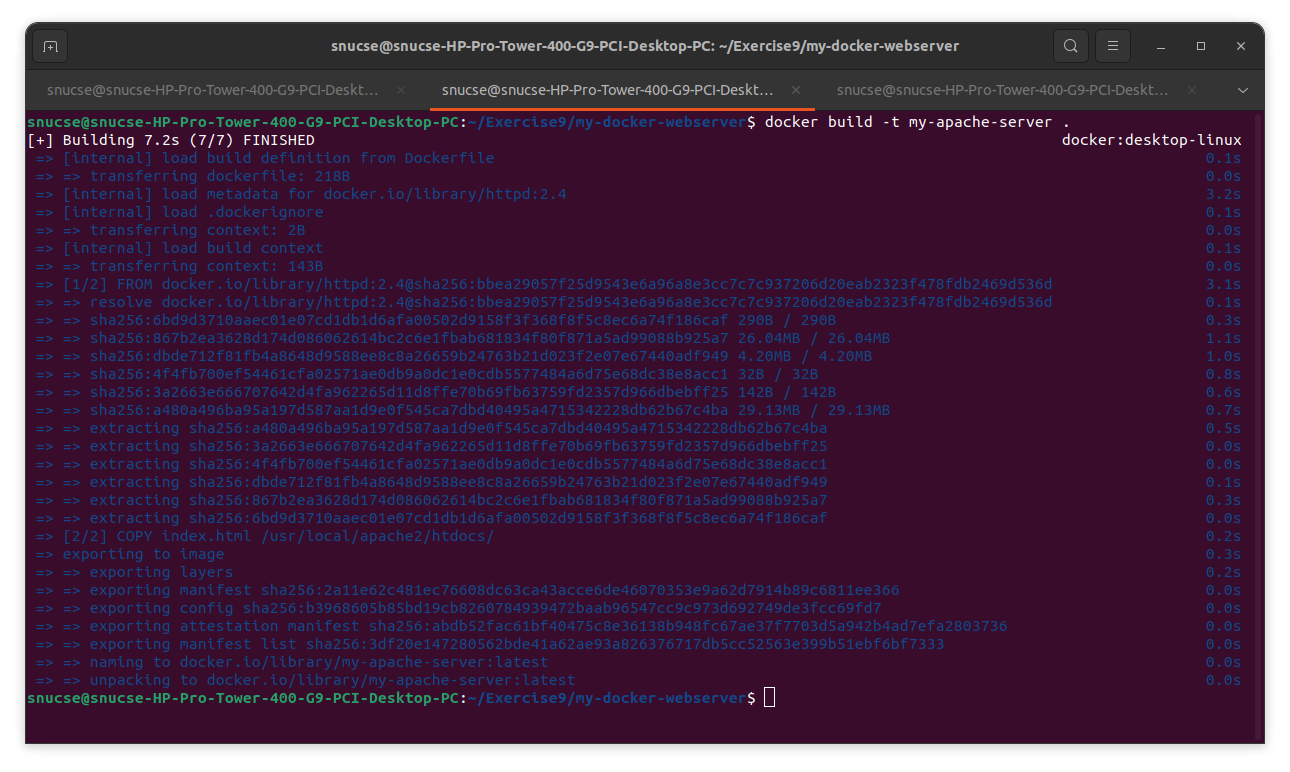
2. Run the following command to build the Docker image:

   docker build -t my-apache-server .

   - **`-t`** flag allows you to tag (name) the image. In this case, the image is named `my-apache-server`.

   - The `.` at the end tells Docker to use the `Dockerfile` and other files from the current directory.

3. You'll see output in the terminal as Docker downloads the Apache image, copies your `index.html` file, and builds the image.



**Step 5: Run the Docker Container**

1. Now, you can run a container from the image you just built. Use the following command:

   docker run -p 8080:80 -d my-apache-server

   - **`-p 8080:80`** maps port 80 in the container (where Apache serves content) to port 8080 on your host machine.

   - **`-d`** runs the container in detached mode, meaning it runs in the background.

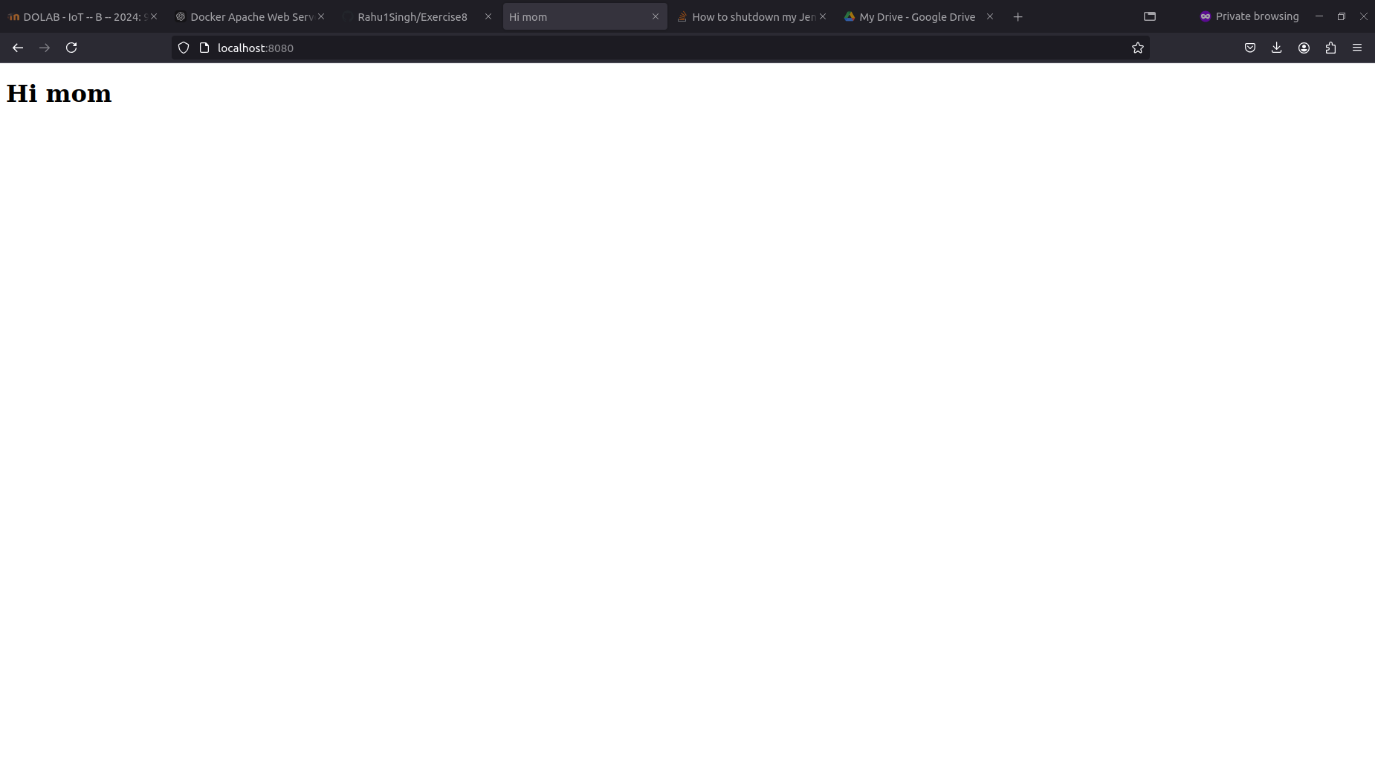
2. After running this command, Docker will output the container ID.

**Step 6: Access Your Apache Web Server**

1. Open a web browser and navigate to:

   http://localhost:8080

   You should see your "Hi mom" message from the `index.html` file served by the Apache web server inside the Docker container.



**Step 7: Cleanup**

Once you're done, you can stop and clean up the Docker container and image.

1. **Stop the running container** by using the following command:

   docker stop <container\_id>

   Replace `<container\_id>` with the actual container ID you got when you ran the container in Step 5 (you can also get it by running `docker ps` to list running containers).

2. **Remove the stopped container**:

   docker rm <container\_id>

3. **Optionally, remove the Docker image**:

   docker rmi my-apache-server

This will free up space on your system by deleting the image you created.

**Result:**

A docker image was created and an Apache server was hosted on it.

|  |  |
| --- | --- |
| **Ex. No. 10** | **CI-CD Pipeline** |
|  |

**Aim:**

To set up and run a CI-CD pipeline with Jenkins, Git, and Docker on Windows.

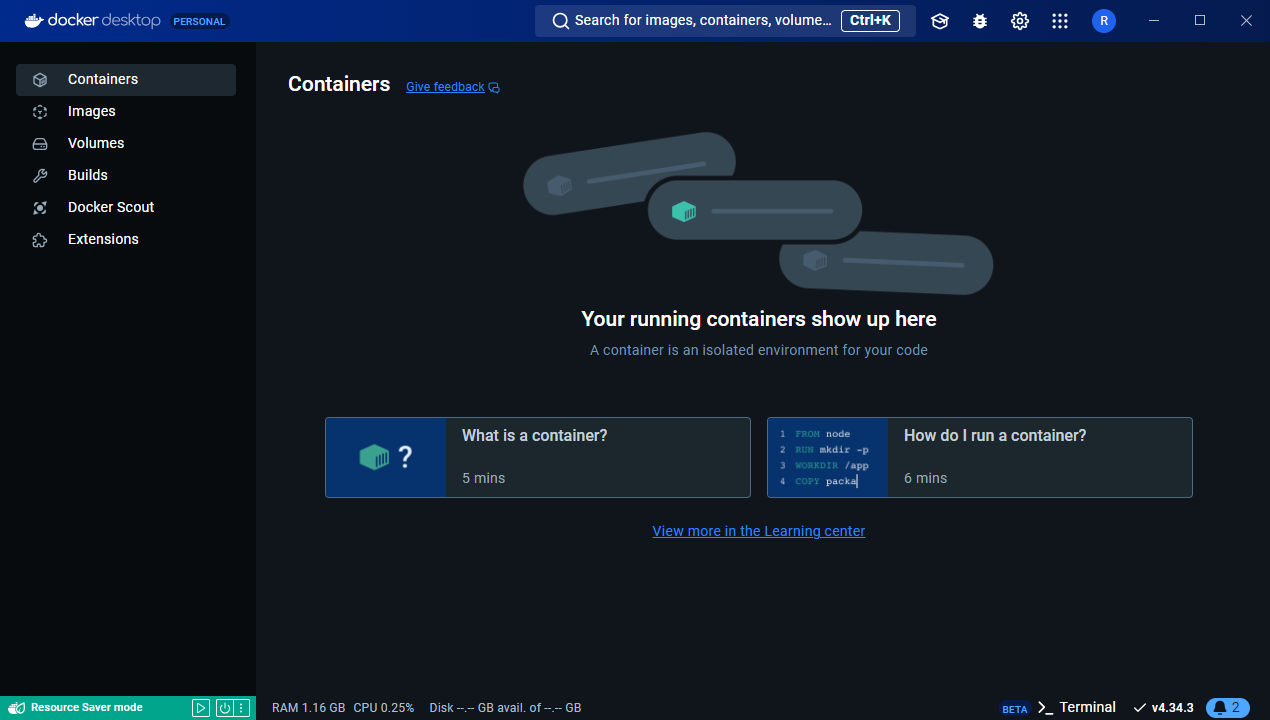
**Algorithm:**

**Step 1: Install Git on Windows**

1. **Download Git** from [git-scm.com](https://git-scm.com/downloads).
2. **Install Git** by running the installer and choosing the recommended settings.
3. **Verify Installation**: Open Command Prompt and type:
4. git --version

**Step 2: Install Docker Desktop for Windows**

1. **Download Docker Desktop** from [docker.com](https://www.docker.com/products/docker-desktop).
2. **Install Docker** by running the installer and enabling Windows containers if prompted.
3. **Enable WSL 2** during installation if you’re using Windows 10 or later.
4. **Verify Installation**:
   * Open PowerShell and type:
   * docker --version
   * Start Docker Desktop, and it will run in the background.

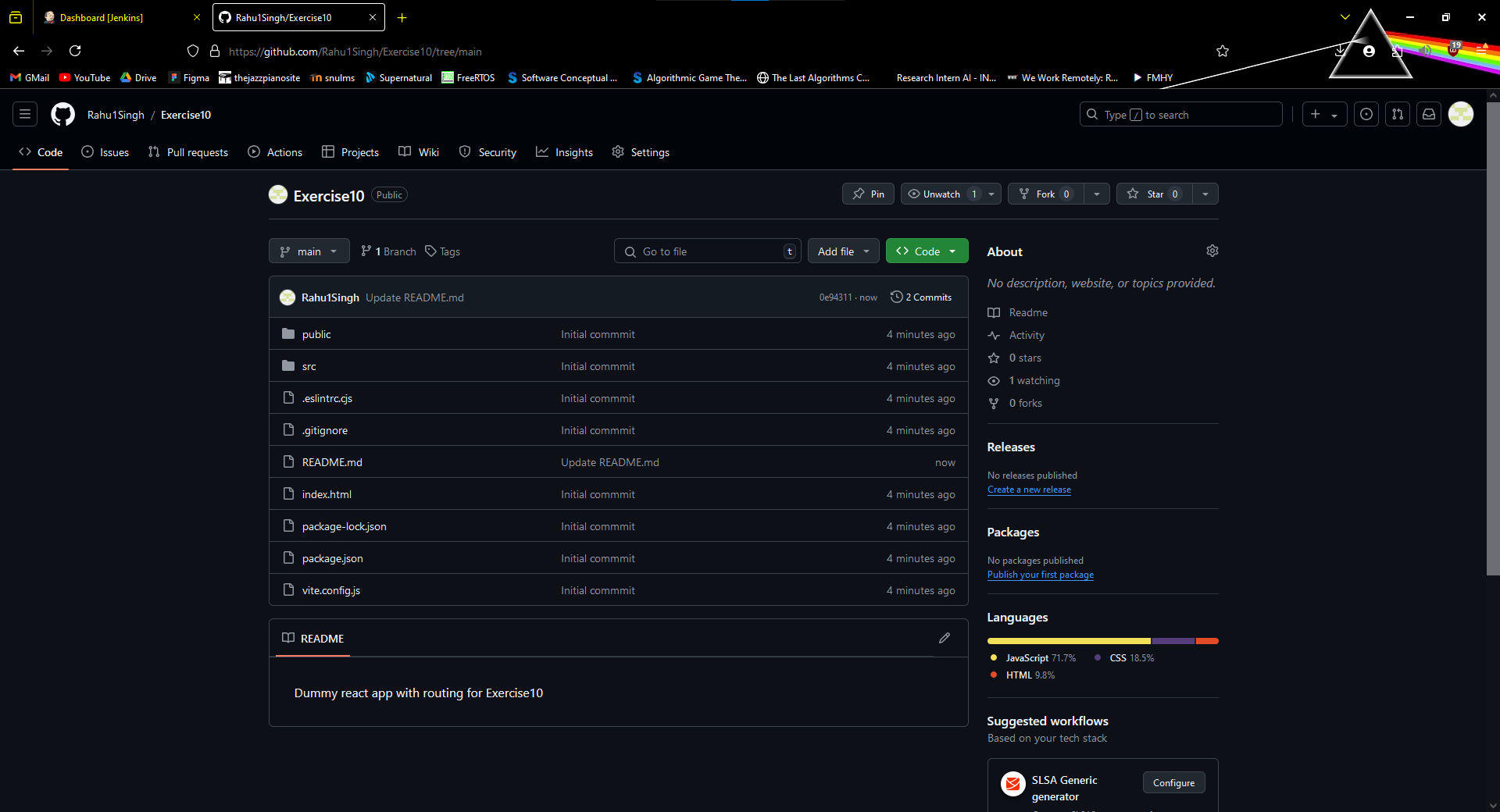
****

**Step 3: Install Jenkins on Windows**

1. **Download Jenkins** from [jenkins.io](https://www.jenkins.io/download/).
2. **Install Jenkins**:
   * Run the installer, following the default options.
   * During setup, Jenkins will prompt for an admin password. Find it in:
   * C:\Program Files (x86)\Jenkins\secrets\initialAdminPassword
   * Open Jenkins in a browser at [http://localhost:8080](http://localhost:8080/) and complete setup.
3. **Install Git and Docker Plugins**:
   * Go to **Manage Jenkins** -> **Manage Plugins** -> **Available**.
   * Install **Git** and **Docker** plugins.

**Step 4: Set Up Your Web Application and Git Repository**

1. **Create a simple web application**:
   * Create a new folder, add a basic index.html or similar.
   * Add a Dockerfile to containerize it. Example:
   * FROM nginx:alpine
   * COPY . /usr/share/nginx/html
2. **Initialize a Git repository**:
   * Open Git Bash, navigate to your project folder, and run:
   * git init
   * git add .
   * git commit -m "Initial commit"
   * git remote add origin <your-git-repo-url>
   * git push -u origin master

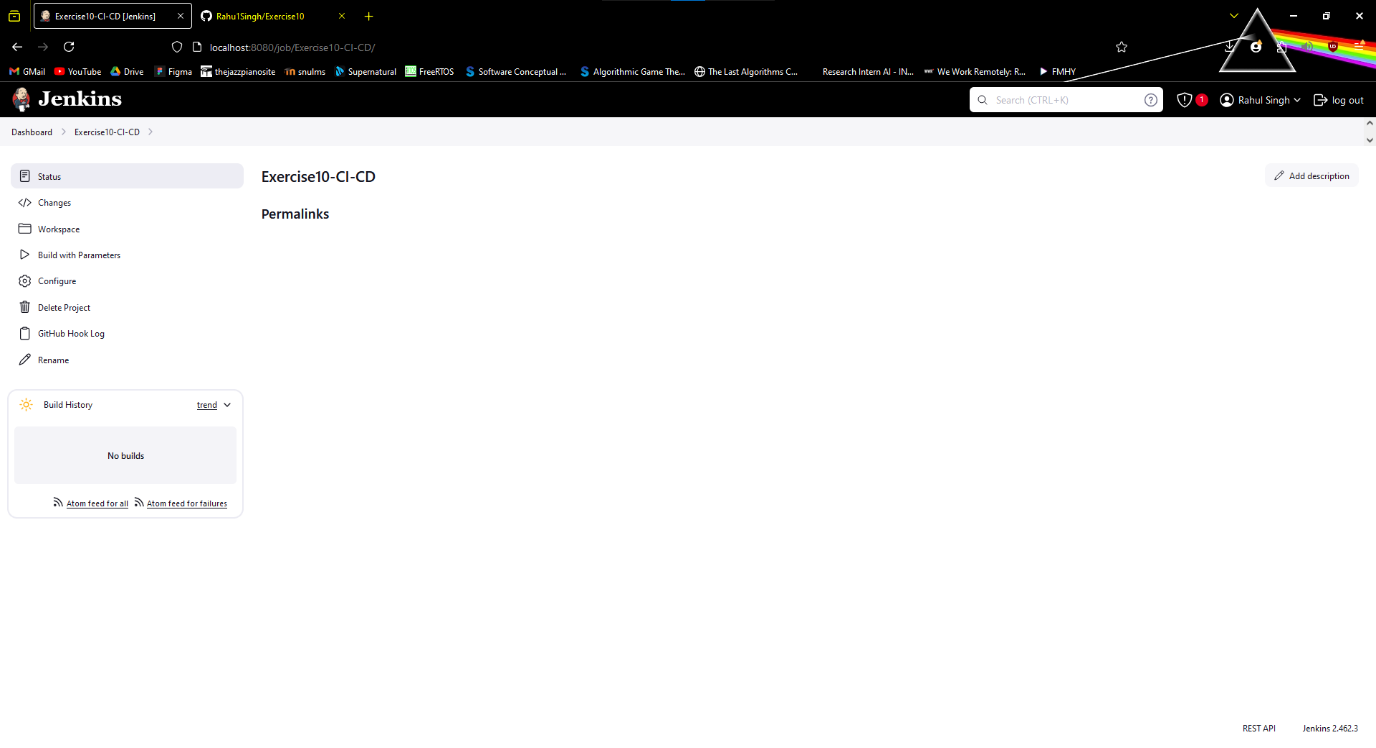
****

**Step 5: Configure Jenkins with Git**

1. Go to **Manage Jenkins** -> **Manage Credentials**.
2. Add your GitHub credentials (SSH key or token) in Jenkins.

**Step 6: Create a Jenkins Job**

1. In Jenkins, select **New Item**, choose **Freestyle project**, and name it (e.g., WebApp-CI-CD).
2. Select **This project is parameterized** and add a **String Parameter** called GIT\_REPO\_URL.
3. In **Source Code Management**, enter your Git repository and set **Branch Specifier** to your working branch (e.g., \*/master).
4. In **Build Triggers**, enable **GitHub hook trigger for GITScm polling**.

****

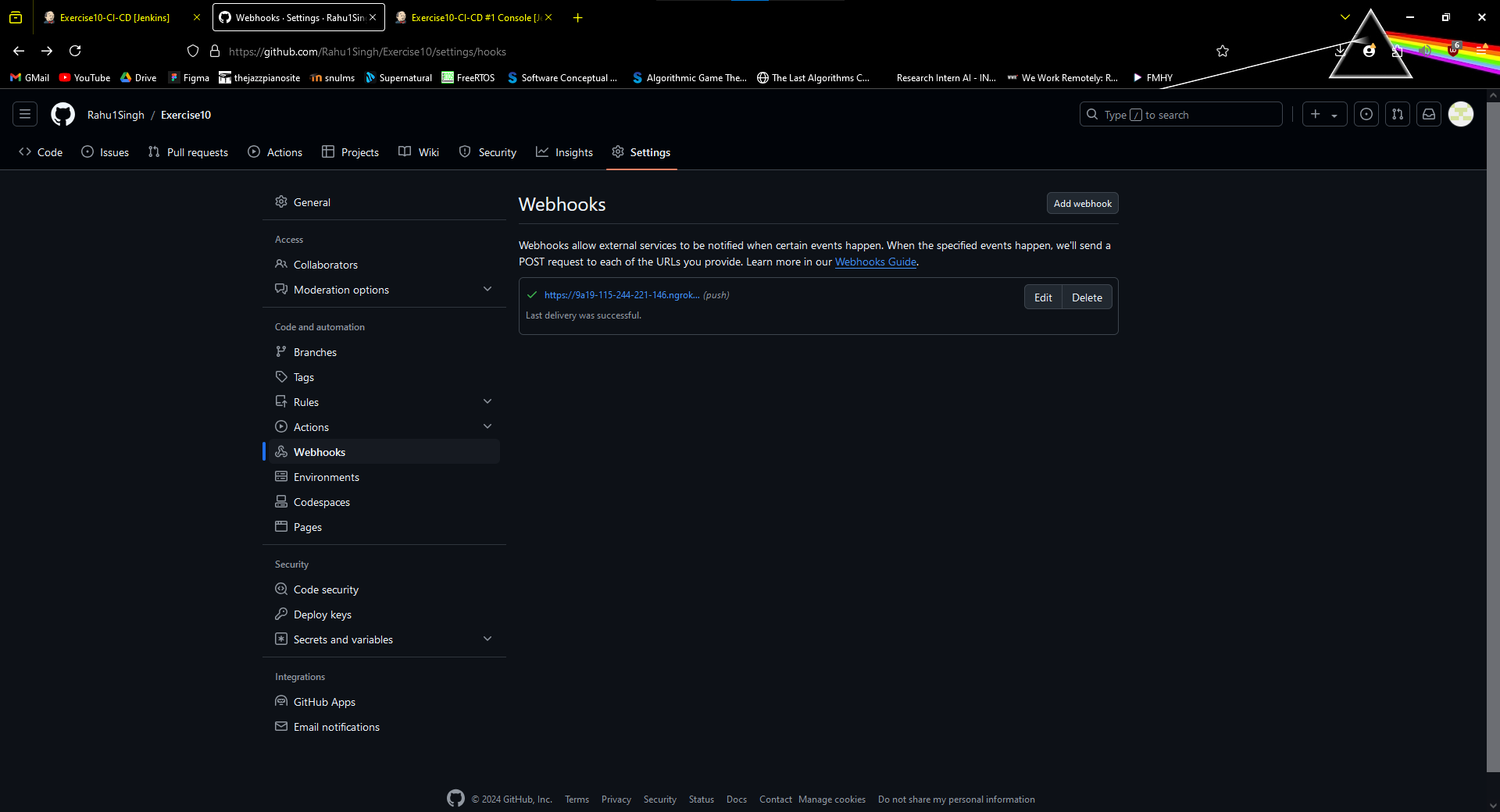
**Step 7: Configure Build Steps**

1. In **Build**, add an **Execute Windows batch command** with these Docker commands:
2. docker rm --force container1 || true
3. docker build -t nginx-image1 .
4. docker run -d -p 8081:80 --name=container1 nginx-image1

This removes any existing container, builds a new Docker image, and runs the container on port 8081.

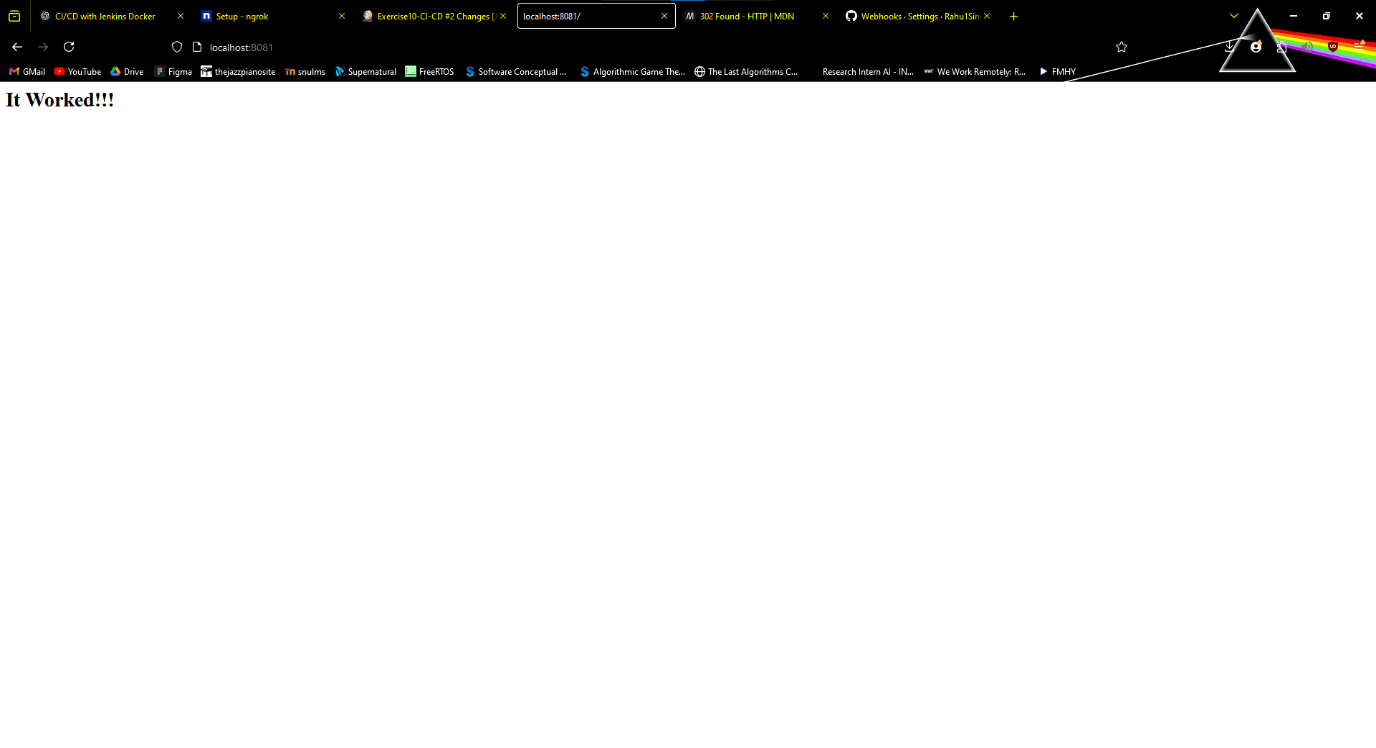
**Step 8: Set Up a GitHub Webhook**

1. Go to your GitHub repository -> **Settings** -> **Webhooks**.
2. Add a new webhook:
   * **Payload URL**: http://<your-jenkins-server>/github-webhook/
   * **Content type**: application/json
   * Set the trigger to "Just the push event."

****

**Step 9: Trigger the CI/CD Pipeline**

1. Push any changes to your GitHub repository:
2. git add .
3. git commit -m "Trigger Jenkins build"
4. git push
5. This triggers the Jenkins job, building and deploying the Docker container.

****

**Result:**

A CI/CD pipeline has been set up for an index.html using Jenkins, Docker, and GitHub.

|  |  |
| --- | --- |
| **Ex. No. 11** | **NodeJS Application** |
|  |

**Aim:**

To set up a CI/CD pipeline for a NodeJS application.

**Procedure:**

**Step 1: Install Git on Windows**

1. **Download Git** from [git-scm.com](https://git-scm.com/downloads).
2. **Install Git** by running the installer and choosing the recommended settings.
3. **Verify Installation**: Open Command Prompt and type:

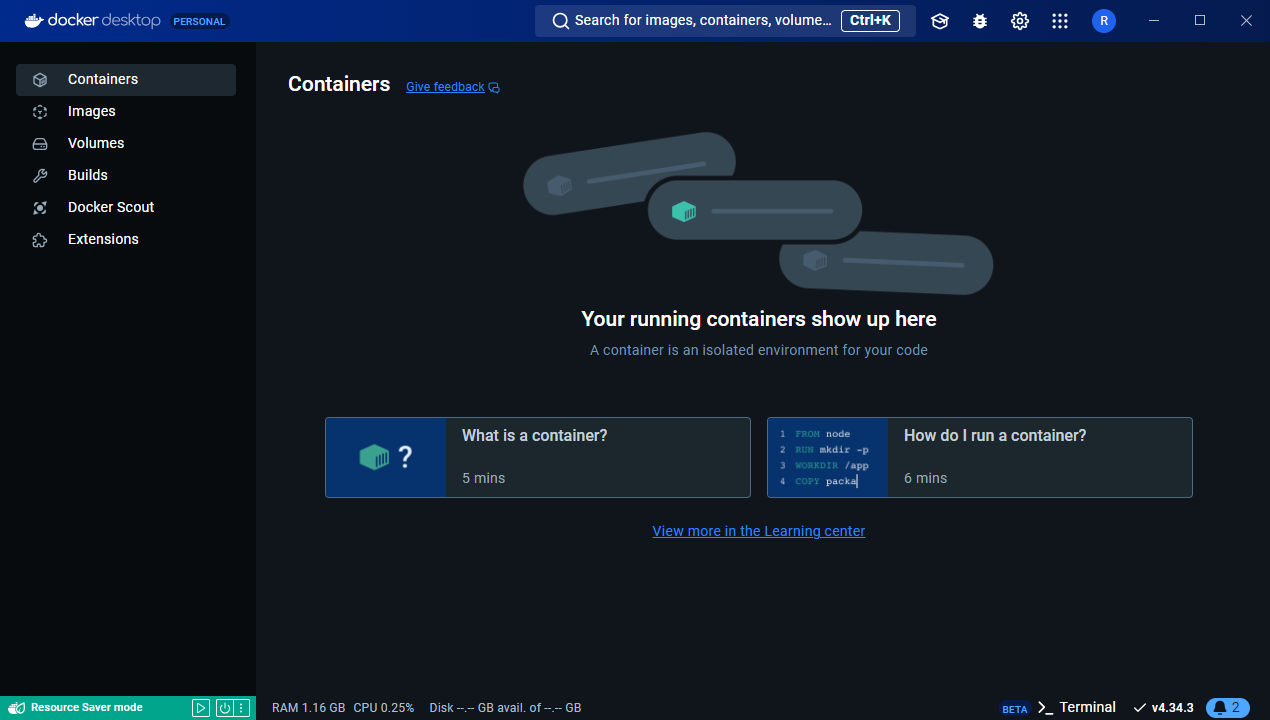
git --version

**Step 2: Install Docker Desktop for Windows**

1. **Download Docker Desktop** from [docker.com](https://www.docker.com/products/docker-desktop).
2. **Install Docker** by running the installer and enabling Windows containers if prompted.
3. **Enable WSL 2** during installation if you’re using Windows 10 or later.
4. **Verify Installation**:
   * Open PowerShell and type:

docker --version

* + Start Docker Desktop, and it will run in the background.



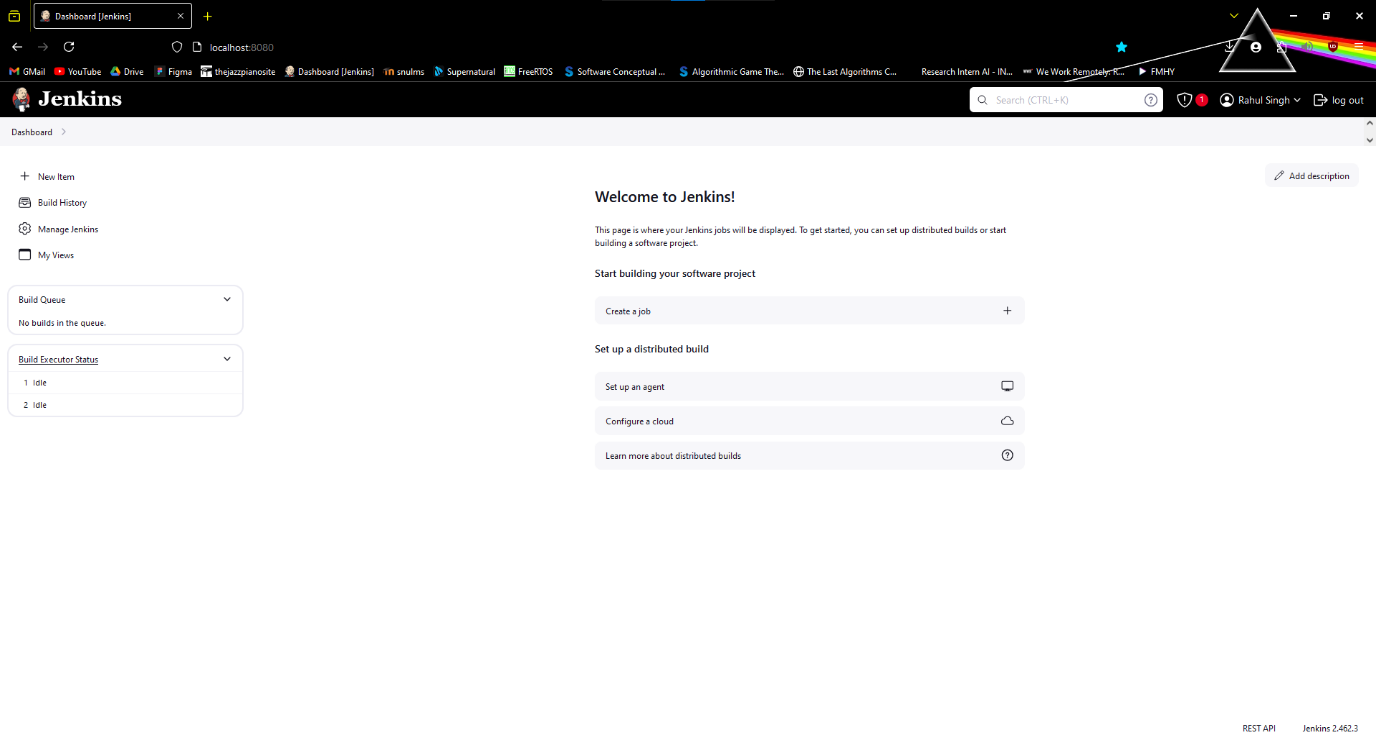
**Step 3: Install Jenkins on Windows**

1. **Download Jenkins** from [jenkins.io](https://www.jenkins.io/download/).
2. **Install Jenkins**:
   * Run the installer, following the default options.
   * During setup, Jenkins will prompt for an admin password. Find it in:

C:\Program Files (x86)\Jenkins\secrets\initialAdminPassword

* + Open Jenkins in a browser at [http://localhost:8080](http://localhost:8080/) and complete setup.

1. **Install Git and Docker Plugins**:
   * Go to **Manage Jenkins** -> **Manage Plugins** -> **Available**.
   * Install **Git** and **Docker** plugins.



**Step 4: Set Up Your Web Application and Git Repository**

1. **Create a simple web application**:
   * Add a simple HTTP server in an index.js file:

const express = require('express');

const app = express();

const port = 3000;

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

res.send('Hello, this is your Node.js application!');

});

app.listen(port, () => {

console.log(`App running on http://localhost:${port}`);

});

* + Add a Dockerfile to containerize it. Example:

FROM nginx:alpine

COPY . /usr/share/nginx/html

1. **Initialize a Git repository**:
   * Open Git Bash, navigate to your project folder, and run:

git init

git add .

git commit -m "Initial commit"

git remote add origin <your-git-repo-url>

git push -u origin master

**Step 5: Configure Jenkins with Git**

1. Go to **Manage Jenkins** -> **Manage Credentials**.
2. Add your GitHub credentials (SSH key or token) in Jenkins.

**Step 6: Create a Jenkins Job**

1. In Jenkins, select **New Item**, choose **Freestyle project**, and name it (e.g., WebApp-CI-CD).
2. Select **This project is parameterized** and add a **String Parameter** called GIT\_REPO\_URL.
3. In **Source Code Management**, enter your Git repository and set **Branch Specifier** to your working branch (e.g., \*/master).
4. In **Build Triggers**, enable **GitHub hook trigger for GITScm polling**.

**Step 7: Configure Build Steps**

1. In **Build**, add an **Execute Windows batch command** with these Docker commands:

docker rm --force container1 || true

docker build -t nginx-image1 .

docker run -d -p 8081:80 --name=container1 nginx-image1

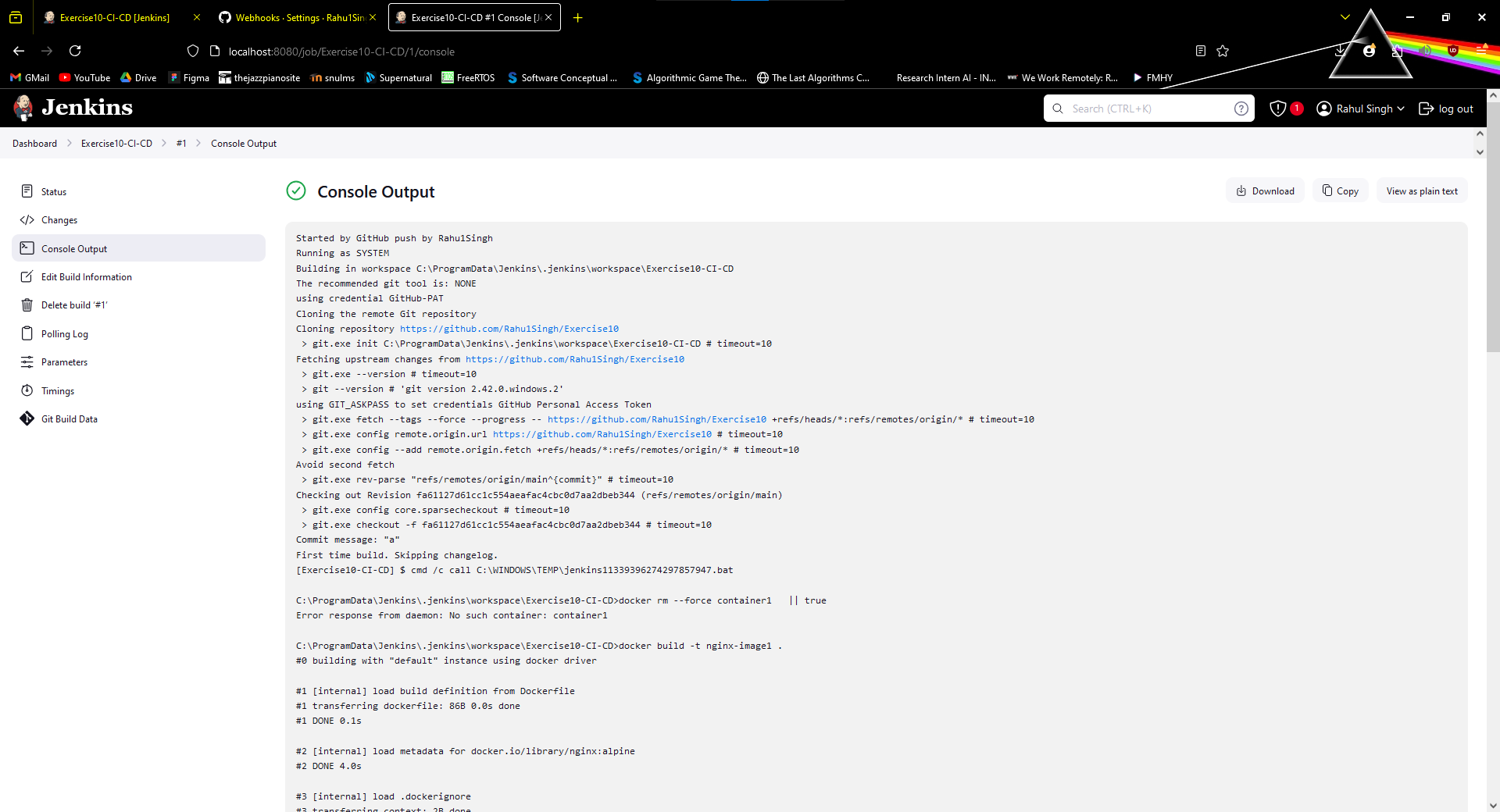
This removes any existing container, builds a new Docker image, and runs the container on port 8081.

**Step 8: Set Up a GitHub Webhook**

1. Go to your GitHub repository -> **Settings** -> **Webhooks**.
2. Add a new webhook:
   * **Payload URL**: http://<your-jenkins-server>/github-webhook/
   * **Content type**: application/json
   * Set the trigger to "Just the push event."

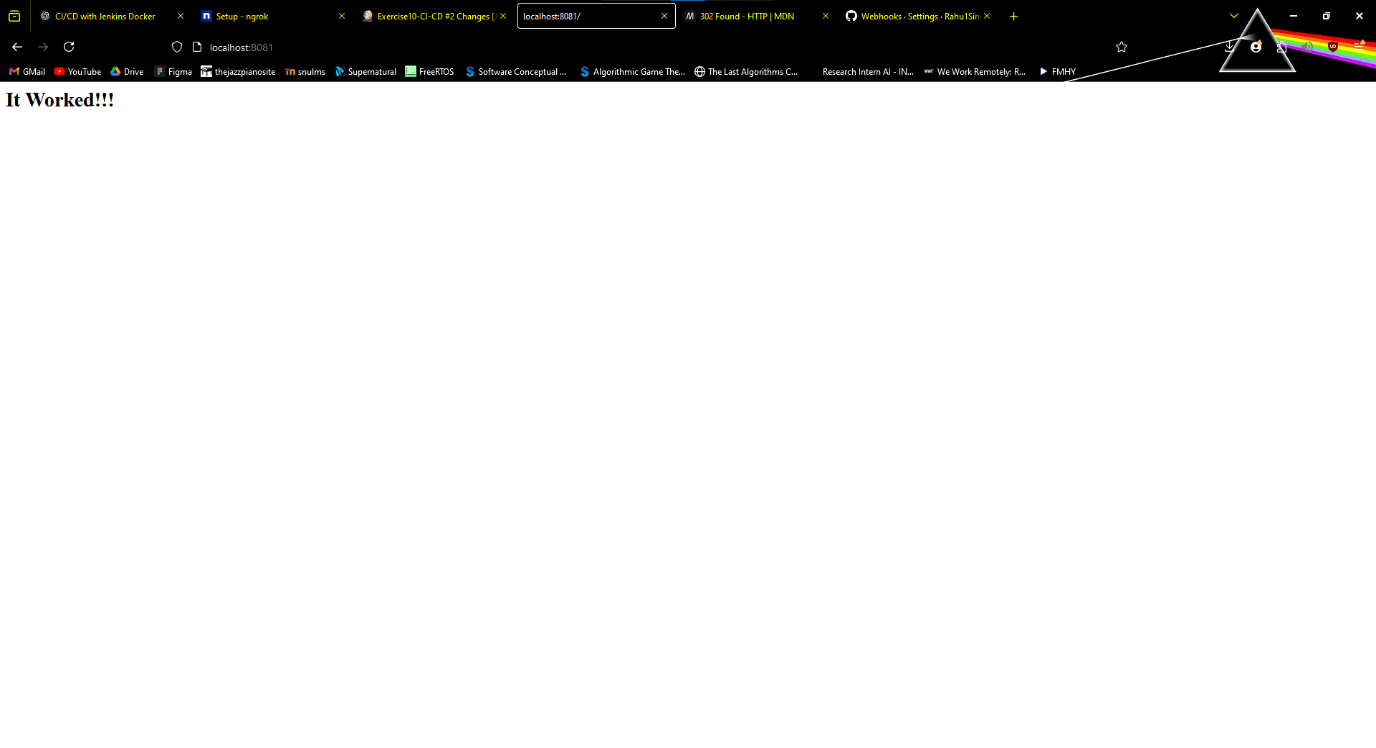
**Step 9: Trigger the CI/CD Pipeline**

1. Push any changes to your GitHub repository:
2. This triggers the Jenkins job, building and deploying the Docker container.



**Step 10: Verify Deployment**

1. Open a browser and go to http://localhost:8081 to view your deployed web application.



**Result:**

A CI/CD pipeline was set up for a NodeJS application using Jenkins, Docker and GitHub Webhooks.

|  |  |
| --- | --- |
| **Ex. No. 12** | **Blue and Green Deployment** |
|  |

**Aim:**

using Jenkins, Git, and Docker on Windows, following the CI/CD principles with a Blue-Green deployment strategy.

**Algorithm:**

**Step 1: Set Up Your Node.js Application**

1. **Create a Node.js project** by navigating to the project directory and initializing it:

mkdir my-node-app

cd my-node-app

npm init -y

1. **Install Express** or any other framework you prefer:

npm install express

1. **Add a simple HTTP server** in an index.js file:

const express = require('express');

const app = express();

const port = 3000;

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

res.send('Hello, this is your Node.js application!');

});

app.listen(port, () => {

console.log(`App running on http://localhost:${port}`);

});

1. **Test your application** by running node index.js and navigating to http://localhost:3000 in your browser.

**Step 2: Dockerize the Node.js Application**

1. In the root of your project, **create a Dockerfile**:

*# Use the official Node.js image*

FROM node:14

*# Create app directory*

WORKDIR /usr/src/app

*# Install app dependencies*

COPY package\*.json ./

RUN npm install

*# Copy app source code*

COPY . .

*# Expose the port the app runs on*

EXPOSE 3000

*# Run the app*

CMD [ "node", "index.js" ]

1. **Build and test the Docker image**:

docker build -t my-node-app .

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

1. **Push the Docker image** to Docker Hub or your preferred container registry for later use in Jenkins.

**Step 3: Configure Jenkins for the Node.js Application**

1. Open Jenkins in your browser.
2. **Create a new job**:
   * Go to the Jenkins dashboard, click on **New Item**.
   * Name the item (e.g., NodeApp-BlueGreen-Deploy), select **Pipeline**, and click **OK**.
3. **Set up Source Code Management**:
   * Under **Source Code Management**, select **Git** and add the URL for your Node.js application’s Git repository.
4. **Add Build Environment**:
   * Configure any required **Node.js or Docker environments**.

**Step 4: Define the Blue-Green Deployment Pipeline in Jenkins**

1. Go to the **Pipeline** section in Jenkins and select **Pipeline script**.
2. Write the pipeline code for Blue-Green deployment using Docker containers:

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git url: 'https://your-repo-url.git', branch: 'main'

}

}

stage('Build Docker Image') {

steps {

script {

docker.build("my-node-app:${env.BUILD\_ID}")

}

}

}

stage('Deploy Green') {

steps {

script {

*// Run the green container on a different port (e.g., 3001)*

sh "docker run -d -p 3001:3000 --name my-node-app-green my-node-app:${env.BUILD\_ID}"

}

}

}

stage('Test Green') {

steps {

script {

*// Add health checks or functional tests*

def response = sh(script: 'curl -s http://localhost:3001', returnStdout: true).trim()

if (response != 'Hello, this is your Node.js application!') {

error "Health check failed!"

}

}

}

}

stage('Switch Traffic to Green') {

steps {

script {

*// Stop the blue container and start routing traffic to green*

sh "docker stop my-node-app-blue || true"

sh "docker rm my-node-app-blue || true"

sh "docker rename my-node-app-green my-node-app-blue"

}

}

}

stage('Clean Up') {

steps {

script {

*// Remove unused images to save space*

sh "docker image prune -f"

}

}

}

}

post {

always {

cleanWs()

}

}

}

* + The pipeline first builds a Docker image of your Node.js app.
  + Then, it deploys the “Green” version (new version) and tests it.
  + If the tests pass, traffic is routed to the new container by stopping the “Blue” container and renaming “Green” to “Blue.”
  + Finally, cleanup removes any dangling Docker images.

A screenshot of a computer

Description automatically generated

**Step 5: Run and Test the Pipeline**

1. Save your pipeline configuration in Jenkins.
2. **Run the pipeline** by clicking on **Build Now**.
3. Jenkins will execute the stages, build the Docker image, deploy the Node.js app to the green environment, test it, and then switch over traffic if all tests pass.

A screenshot of a computer

Description automatically generated

**Step 6: Validate the Deployment**

1. Access the running application at http://localhost:3001 to confirm the Blue-Green deployment worked.
2. Make any necessary adjustments, especially if there are additional requirements for health checks or tests for your application.

A screen shot of a computer

Description automatically generated

**Result:**

A Node.js application has been successfully deployed using a Blue-Green strategy.