|  |  |
| --- | --- |
| **Ex. No: 12** | Blue Green Deployment |
| **dd.mon.year** |

**Aim:** Implement a Blue-Green Deployment strategy for a Node.js application

**Algorithm:**

1. **The goal is to containerize the app using Docker, push it to Docker Hub, and set up a Jenkins pipeline that deploys the app in two separate environments, blue and green. The deployment should ensure zero-downtime, allowing users to continue using the application while a new version is deployed and tested.**
2. **Create a Pipeline project**
3. **Add the Pipeline Script**
4. **Save Config**
5. **Open ngrok Tunnel to expose 8080 -> Jenkins server and add github webhook**
6. **Set up server using node and push the code using git /SCM**
7. Install Docker / Blue Ocean Plugin in Jenkins
8. Create Docker Image by adding docker file
9. Tag the docker Image Push the Image to docker hub
10. **Trigger the Pipeline**
11. **The Pipeline will pull the code and build the docker img and push to hub**
12. **Deploy to blue green ensuring 0 downtime .**
13. **Access localhost:3001/3002 to see blue-green env deployment**

**Input :**

pipeline {

agent any

triggers {

pollSCM('\* \* \* \* \*') // Poll SCM every minute for changes

}

stages {

stage('Clone Repository') {

steps {

// Cloning the Git repository

checkout([$class: 'GitSCM', branches: [[name: '\*/main']],

userRemoteConfigs: [[url: 'https://github.com/Ajay-code-unix/blue-green1.git']]

])

}

}

stage('Build Docker Image') {

steps {

script {

sh "sudo docker build --no-cache -t ajaybadrinath/bg:latest ."

}

}

}

stage('Push Docker Image') {

steps {

sh "echo “pwd”| sudo -S docker login -u ajaybadrinath --password-stdin"

sh 'sudo docker push ajaybadrinath/bg:latest'

}

}

stage('Deploy to Blue or Green') {

steps {

script{

// sh 'sudo docker rm -f blue || true'

//sh 'sudo docker run -d --name blue -p 3001:3000 ajaybadrinath/bg:latest'

def env=sh(script: "curl -s http://localhost:3001 || echo green", returnStdout: true).trim()

if (env == "Hello bg") {

deployToGreen()

} else {

deployToBlue()

}

}

}}

}

}

def deployToBlue() {

sh 'sudo docker rm -f blue || true'

sh 'sudo docker run -d --name blue -p 3001:3000 ajaybadrinath/bg:latest'

}

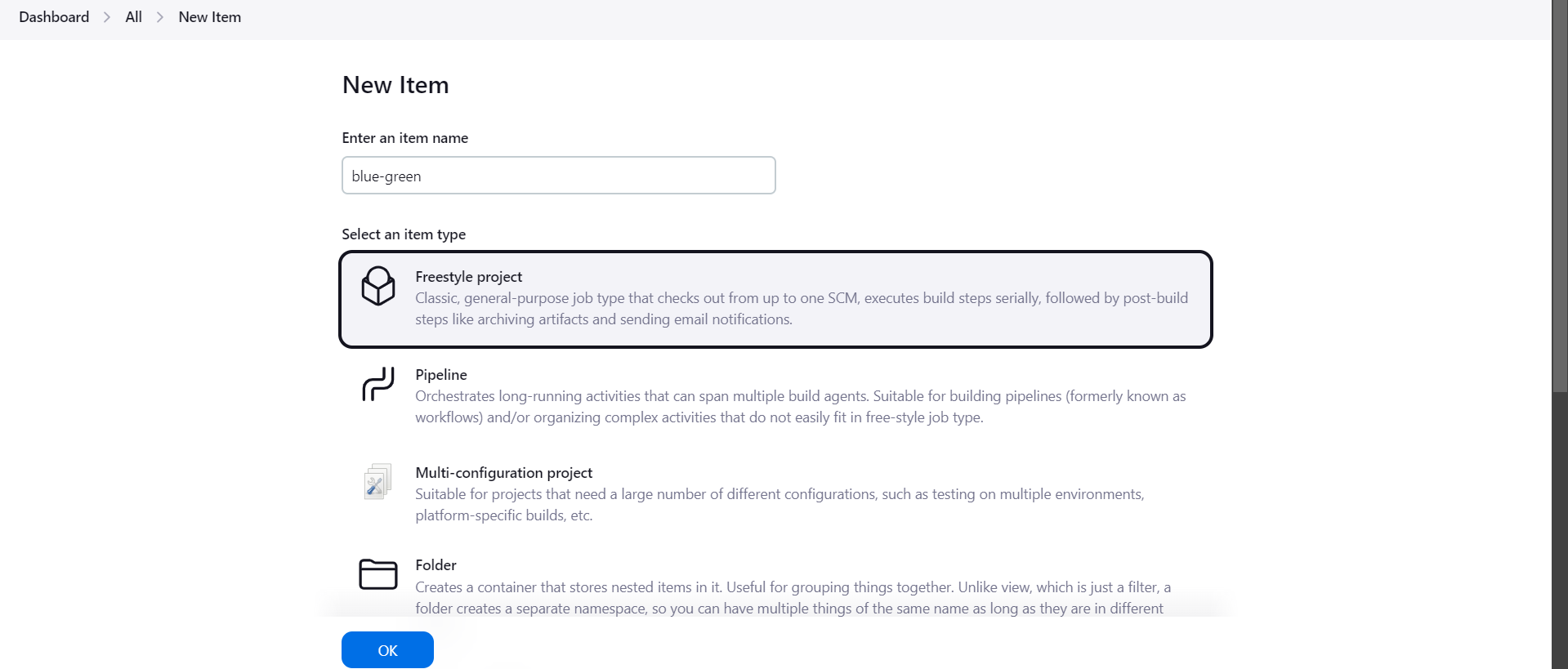
def deployToGreen() {

sh 'sudo docker rm -f green || true'

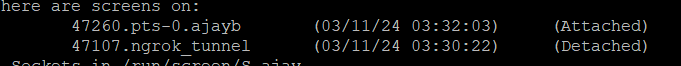
sh 'sudo docker run -d --name green -p 3002:3000 ajaybadrinath/bg:latest'

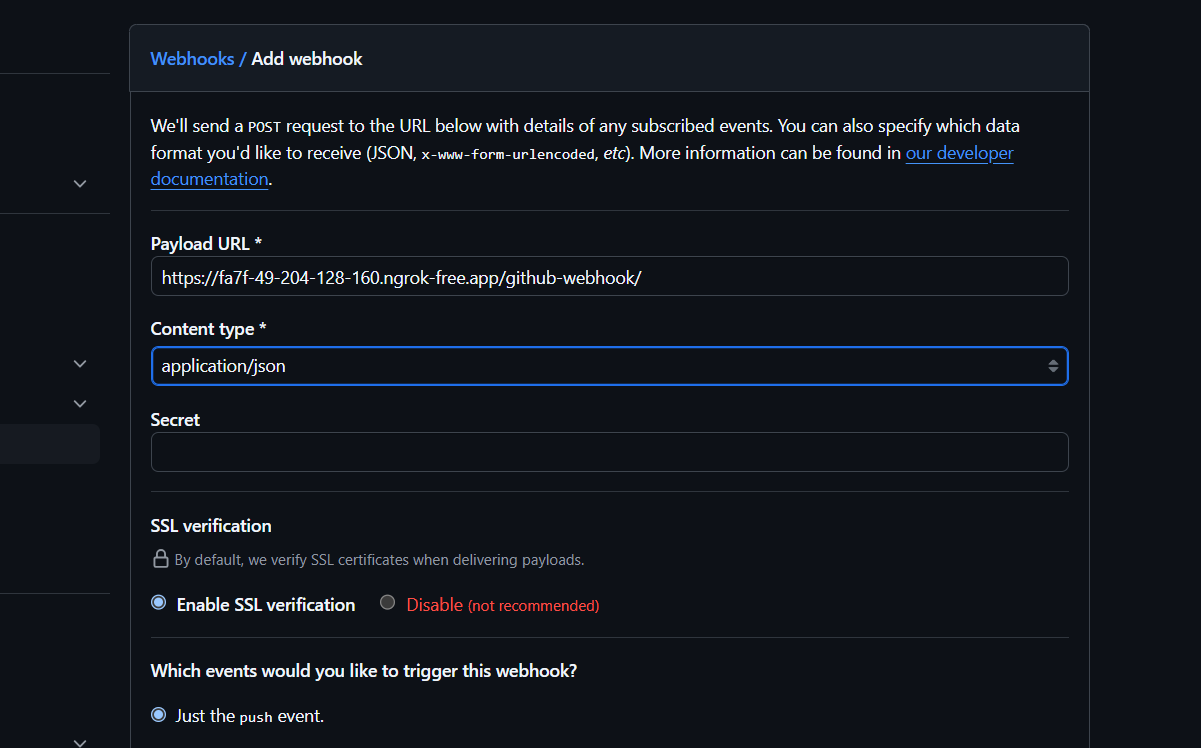
}

**Output: -**

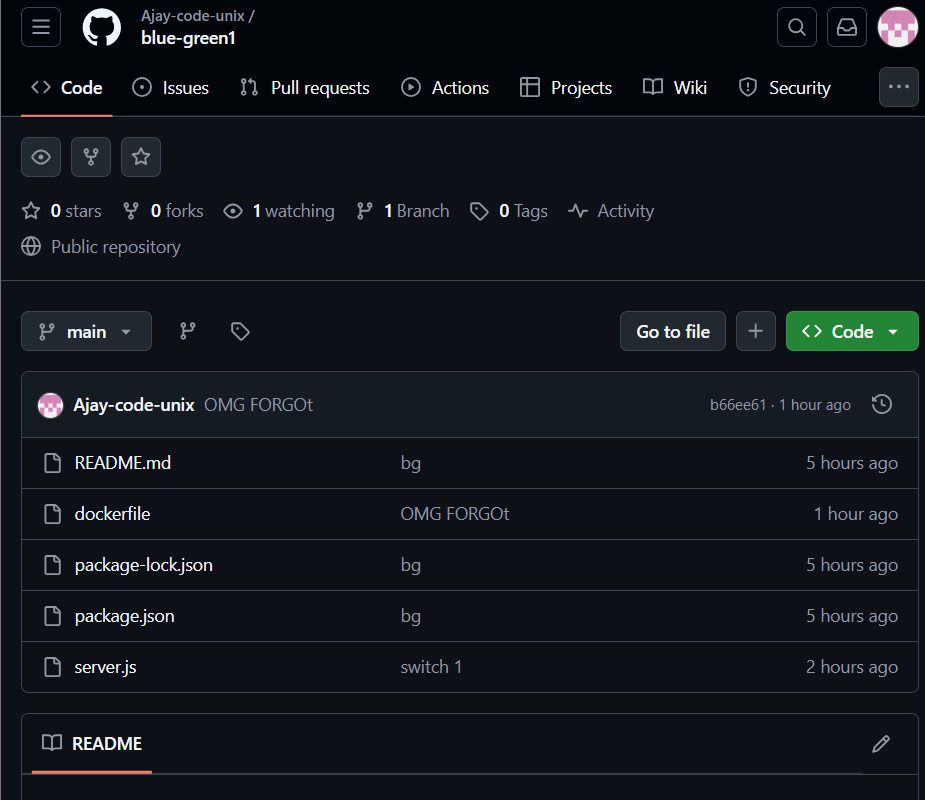


**Create Pipeline**

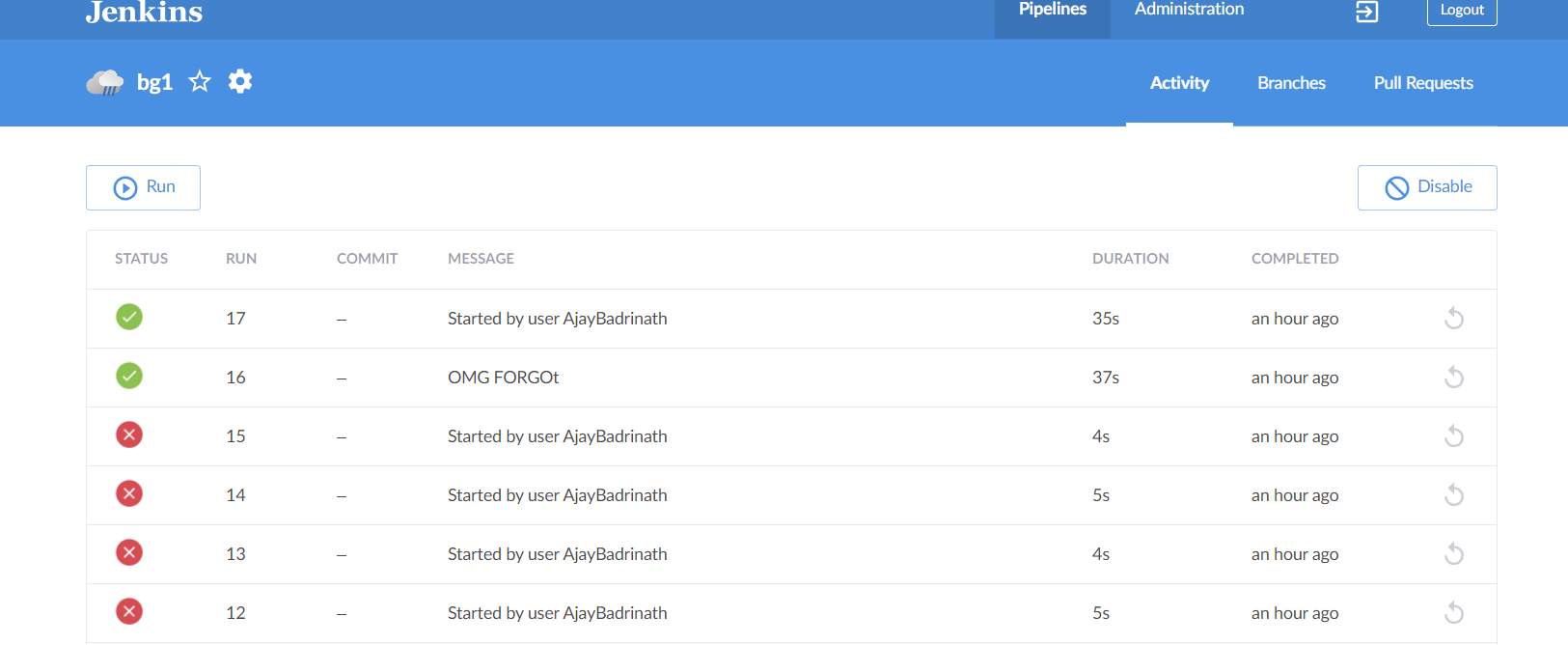
**Open ngrok Tunnel to expose local Jenkins server & Add Webhook**



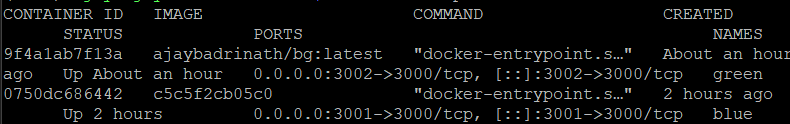
Enable GIT/SCM to manage Source Code



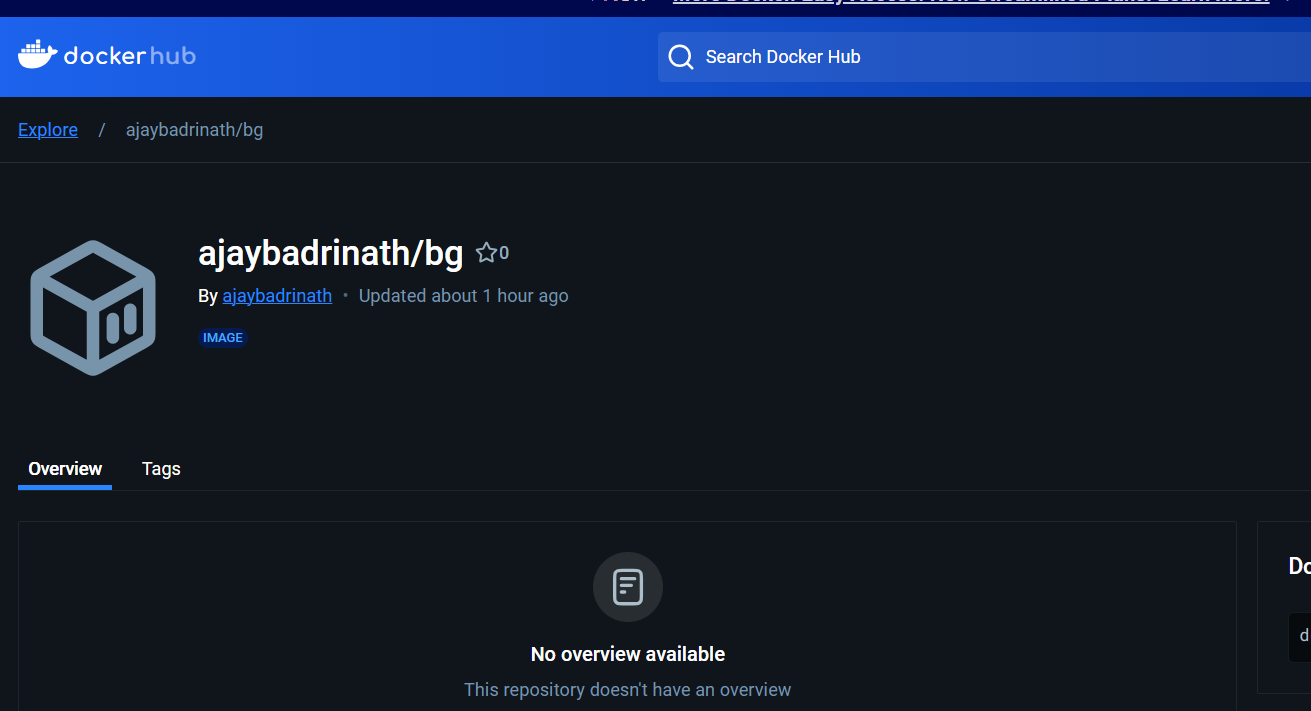
Installed Blue ocean plugin for Jenkins



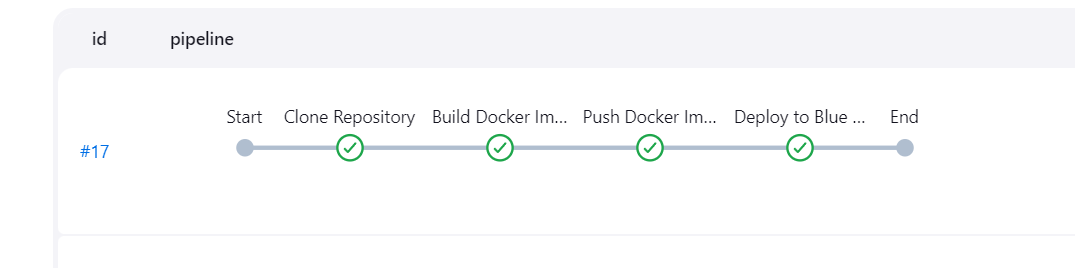
Create Docker Image



Tag the docker image



Pipeline Overall



BLUE ENV :



GREEN ENV:



**Result:** Thus Blue-Green Deployment was done using Jenkins BlueOcean Plugin and Docker .

|  |  |
| --- | --- |
| **Ex. No: 11** | **Applying CI/CD Principles for NodeJS application Using Jenkins, Git, using Docker Containers** |
| **dd.mon.year** |

**Aim:** To Develop a REST API using Node.js. The API should provide an endpoint /status that returns a simple JSON response with a status message and Containerize the API using Docker and automate the entire CI/CD pipeline using Jenkins

**Algorithm:**

1. Add Git Repo Named Lab11 with necessary files such as dockerfile, index.js and package\*.json which are required to build the image
2. Now Once the git repo is created then set up Jenkins with the appropriate configuration
3. From the Jenkins side
   1. Install Jenkins on your computer or server following the instructions for your operating system (https://www.jenkins.io/download/).
   2. Open Jenkins in your web browser (usually at http://localhost:8080) and
   3. complete the initial setup, including setting up an admin user and installing necessary plugins.
   4. Configure Jenkins to work with Git by setting up Git credentials in the Jenkins
   5. Set Branches to build -> Branch Specifier to the working Git branch (ex\*/master)
   6. In the job configuration, go to the "Build Triggers" section and select the
   7. "GitHub hook trigger for GITScm polling" option. This enables Jenkins to listen for GitHub webhook triggers.
4. In the job configuration, go to the "Build" section.
5. Add build steps to execute Docker commands for building and deploying the containerized web application
6. Push changes to your GitHub repository. The webhook will trigger the Jenkins

job automatically, executing the build and deployment steps defined in the job

configuration.

.

Input :

1. Create a simple node server

const express=require('express')

const app = express();

const PORT=3000;

app.get('/',(req,res)=> res.send('Hello 1 There!!!'));

app.listen(PORT,()=>console.log(`APP running on Port ${PORT}`));

2. Create Docker Image

# Use Node.js base image

FROM node:14

# Set the working directory inside the container

WORKDIR /usr/src/app

# Copy package files and install dependencies

COPY package\*.json ./

RUN npm install

# Copy the rest of the application files

COPY . .

# Expose the app's port

EXPOSE 3000

# Command to run the application

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

3. Jenkins Script

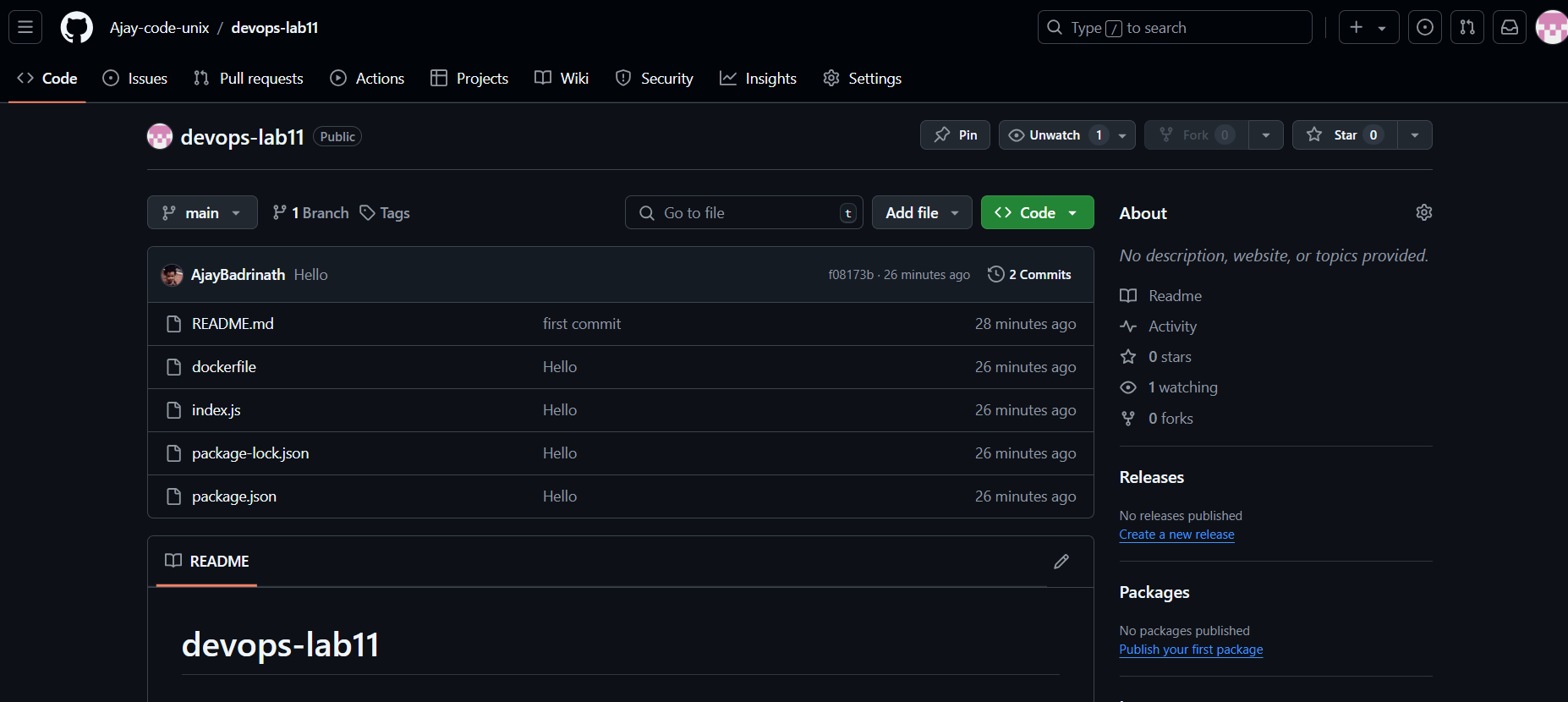
docker rm --force container1 || true

docker build -t node-image .

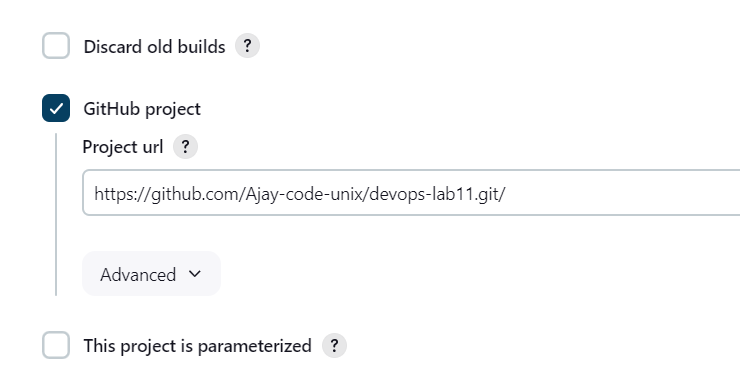
docker run -d -p 3002:3000 --name=container1 node-image

Output:

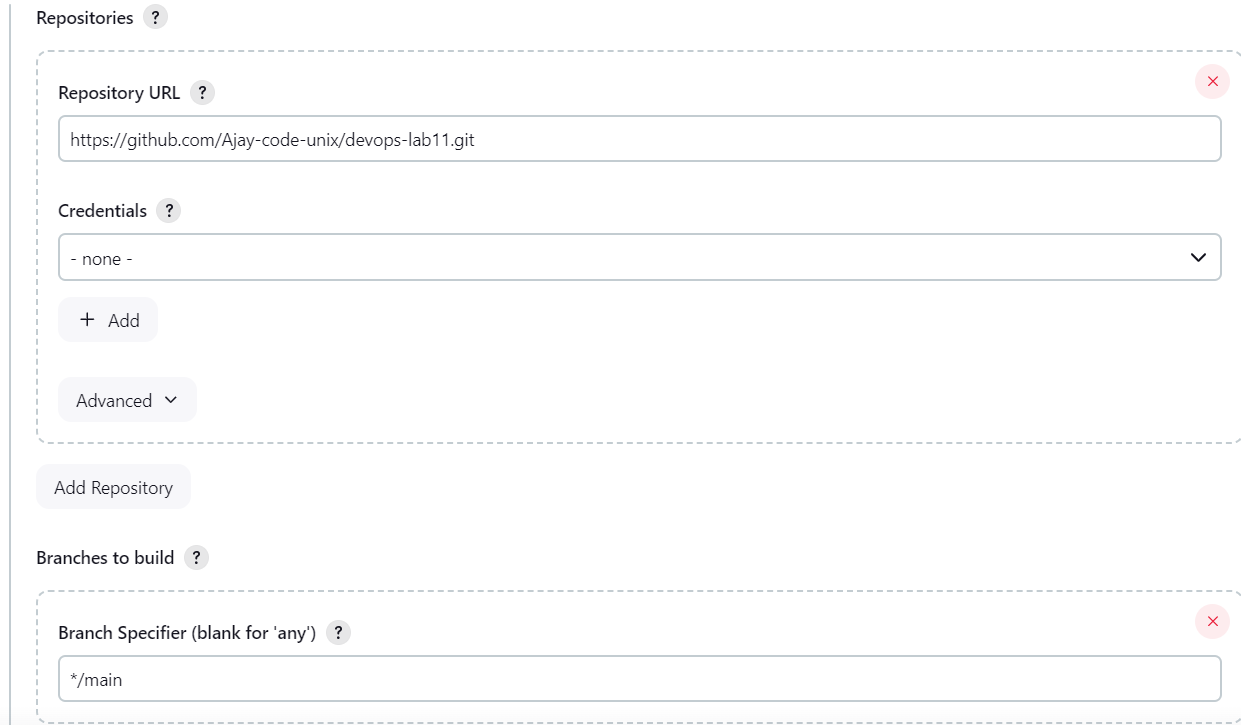
Adding Git repo with docker file and Node index.js

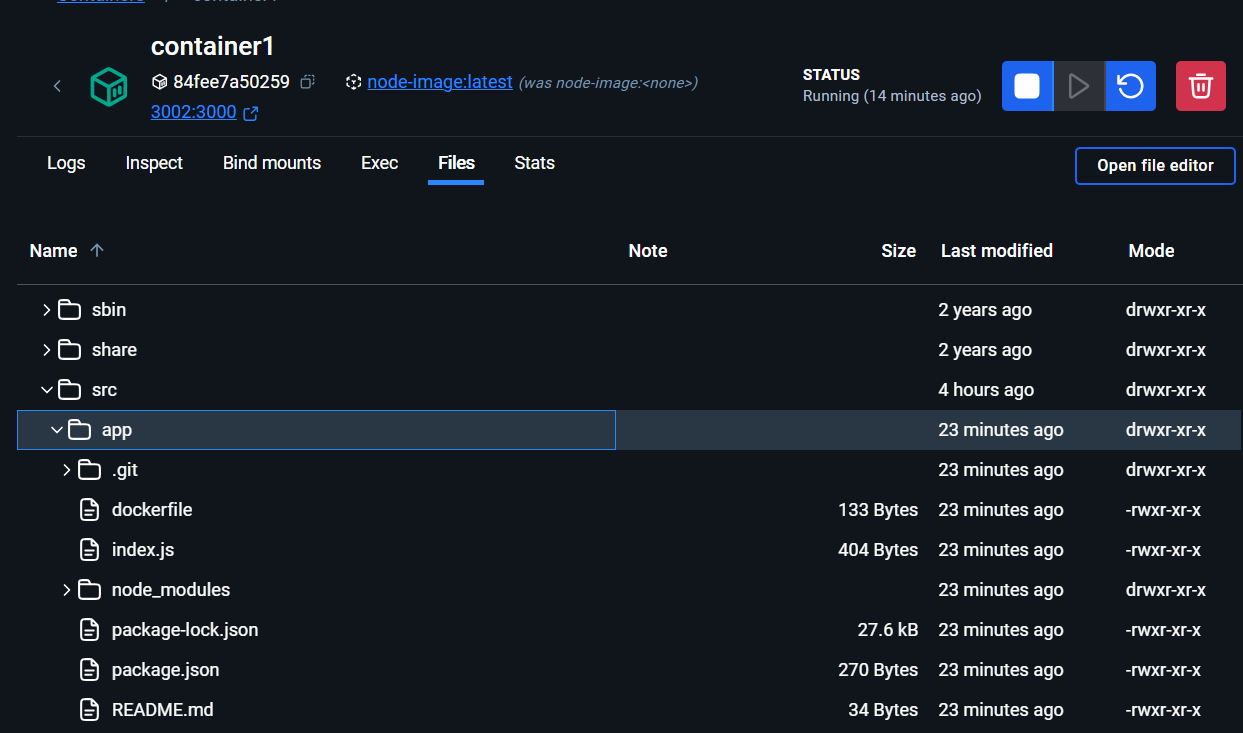


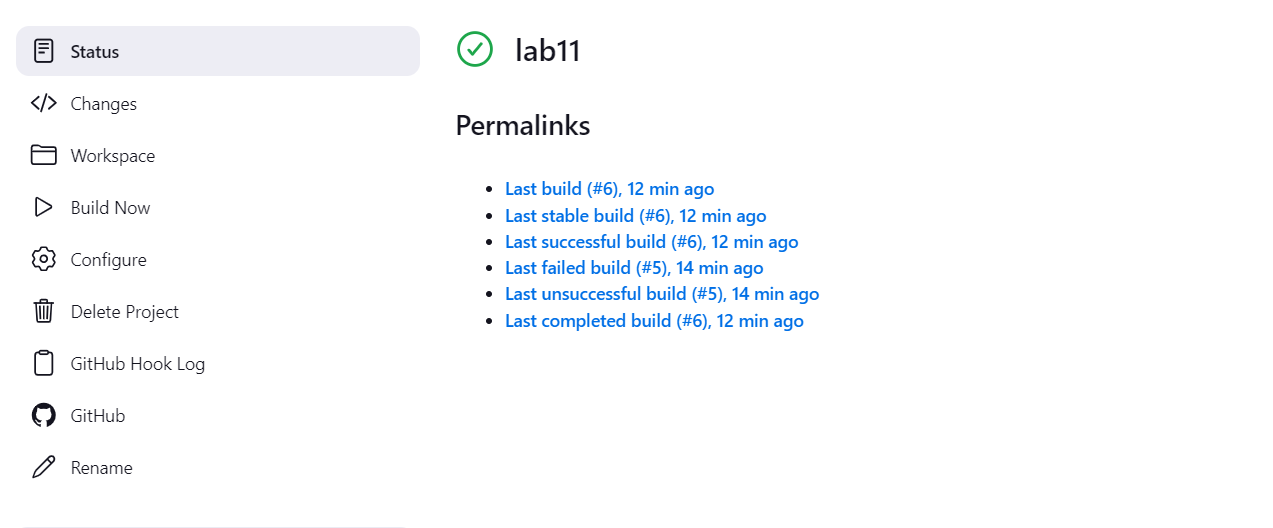
Configuring Jenkins with git credentials



Setting Up Branches to Build







**Result:** Thus CI/CD Principles for Node JS Application was applied using Jenkins+Docker and GIT.

|  |  |
| --- | --- |
| **Ex. No: 10** | **Applying CI/CD Principles for NodeJS application Using Jenkins, Git, using Docker Containers** |
| **dd.mon.year** |

**Aim: To Apply CI-CD Principles using Docker containers and Git**

**Procedure :**

**Step 1: Set Up the Web Application and Git Repository**

Create a simple web application or use an existing one. Ensure it can be

hosted in a Docker container.

Initialise a Git repository for your web application and push it to GitHub.

**Step 2: Install and Configure Jenkins**

Install Jenkins on your computer or server following the instructions for your operating system (https://www.jenkins.io/download/).

Open Jenkins in your web browser (usually at http://localhost:8080) and

complete the initial setup, including setting up an admin user and installing

necessary plugins.

Configure Jenkins to work with Git by setting up Git credentials in the Jenkins

Credential Manager.

**Step 3: Create a Jenkins Job**

* Create a new Jenkins job using the "Freestyle project" type.
* In the job configuration, specify a name for your job and choose "This project is parameterized."
* Add a "String Parameter" named GIT\_REPO\_URL and set its default value to your Git repository URL.
* Set Branches to build -> Branch Specifier to the working Git branch (ex\*/master)
* In the job configuration, go to the "Build Triggers" section and select the
* "GitHub hook trigger for GITScm polling" option. This enables Jenkins to
* listen for GitHub webhook triggers.

**Step 4: Configure Build Steps**

In the job configuration, go to the "Build" section.

Add build steps to execute Docker commands for building and deploying the

containerized web application. Use the following commands:

**# Remove the existing container if it exists**

docker rm --force container1

**# Build a new Docker image**

docker build -t nginx-image1 .

**# Run the Docker container**

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

These commands remove the existing container (if any), build a Docker

image named "nginx-image1," and run a Docker container named

"container1" on port 8081.

**Step 5: Set Up a GitHub Webhook**

In your GitHub repository, navigate to "Settings" and then "Webhooks."

[Create a new webhook, and configure it to send a payload to the Jenkins

webhook URL (usually http://jenkins-server/github-webhook/). Set the content

type to "application/json."

**Step 6: Trigger the CI/CD Pipeline**

Push changes to your GitHub repository. The webhook will trigger the Jenkins

job automatically, executing the build and deployment steps defined in the job

configuration.

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

**Step 7: Verify the Deployment**

Access your web application by opening a web browser and navigating to

http://localhost:8081 (or the appropriate URL if hosted elsewhere).

**Input :**

**Create a Docker file for apache server**

**FROM httpd:2.4**

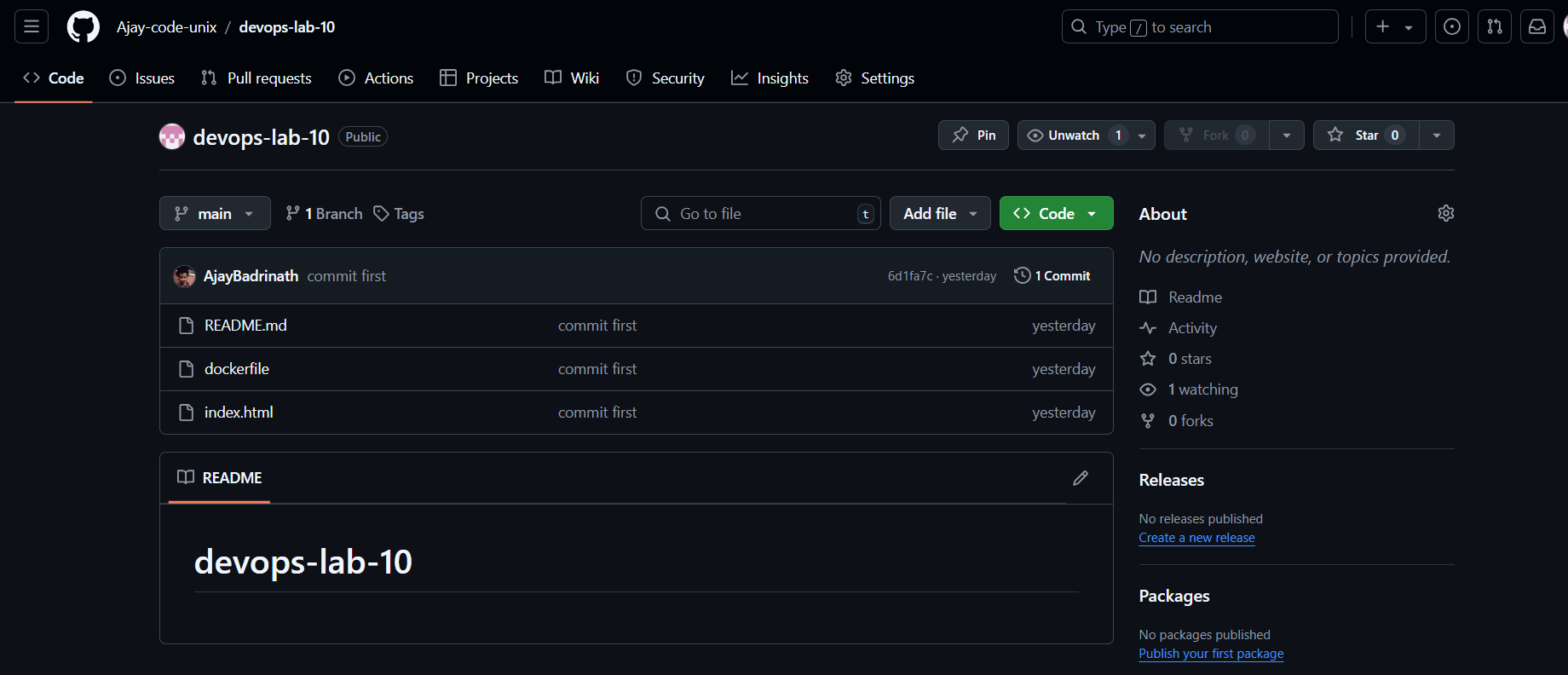
**COPY index.html /usr/local/apache2/htdocs/**

**Create a simple HTML Static site to serve**

**<p> hello docker </p>**

**Output:**

**Initializing Git repo and Push to GH**



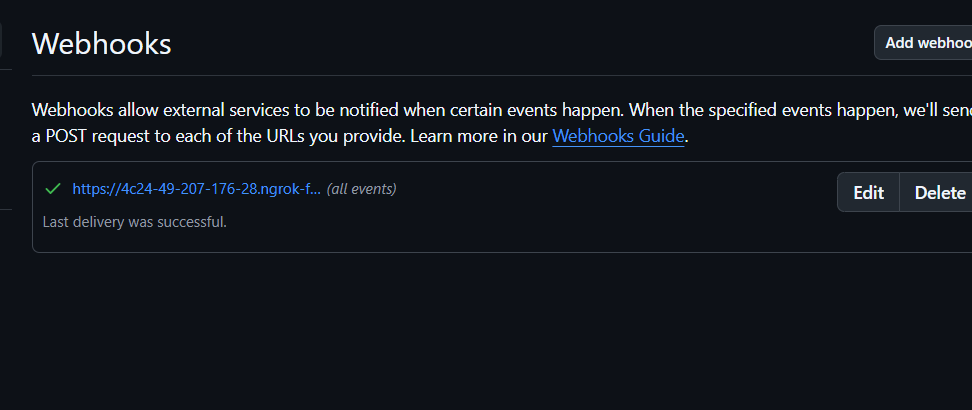
Creating Jenkins Job and Configuring



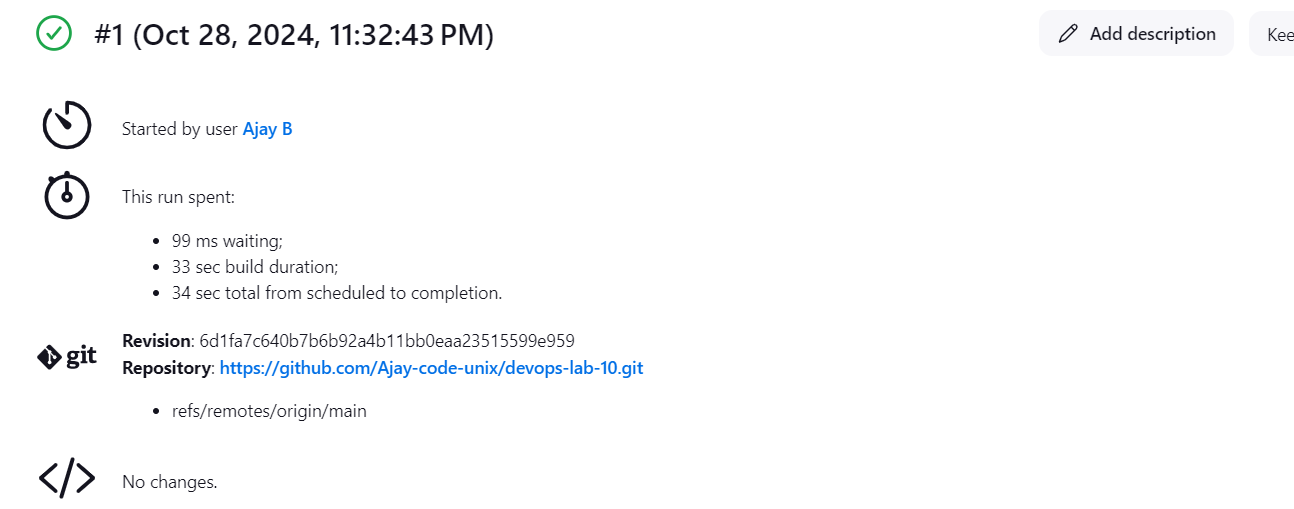
Configure Build Steps



Setup GH Webhook



Trigger the Pipeline



**Result : Thus CI-CD Principles were applied using Jenkins Docker and Git**

|  |  |
| --- | --- |
| **Ex. No: 9** | **Exploring Containerization and Application Deployment with Docker** |
| **dd.mon.year** |

**Aim : To containerize a html application with apache server**

**Procedure:**

**Step 1: Install Docker**

Install Docker on your computer by following the instructions provided on the Docker website (https://docs.docker.com/get-docker/).

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

Create a directory for your web server project.

Inside this directory, create a file named index.html with a simple "Hello, Docker!" message. This will be the content served by your Apache web server.

**Step 3: Create a Dockerfile**

Create a Dockerfile in the same directory as your web server project. The Dockerfile defines how your Apache web server application will be packaged into a Docker container.

**Step 4: Build the Docker Image**

Build the Docker image by running the following command in the same directory as

your Dockerfile:

**Step 5: Run the Docker Container**

Start a Docker container from the image you built:

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

Access your Apache web server by opening a web browser and navigating to

http://localhost:8080. You should see the "Hello, Docker!" message served by your Apache

web server running within the Docker container.

**Step 7: Cleanup**

Stop the running Docker container:

.

Optionally, remove the container and the Docker image:

***docker rm <container\_id>***

***docker rmi my-apache-server***

**Input:**

**Dockerfile**

***Use an official Apache image as the base image***

***FROM httpd:2.4***

***# Copy your custom HTML page to the web server's document root***

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

**Docker Commands**

***docker build -t my-apache-server .***

Replace my-apache-server with a suitable name for your image.

25

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

This command maps port 80 in the container to port 8080 on your host machine and

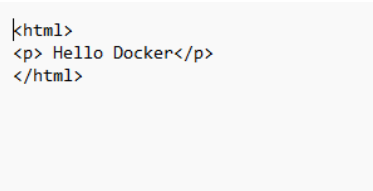
runs the container in detached mode.

**docker stop <container\_id>**

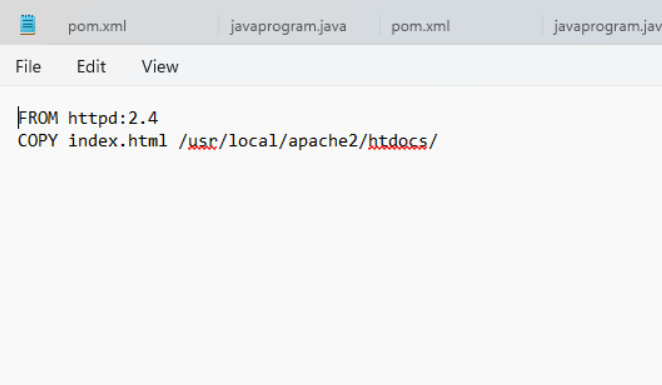
Replace <container\_id> with the actual ID of your running container

Output:

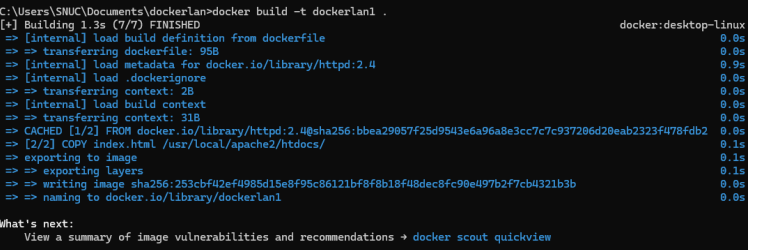
Create Html Page

****

**Docker File Creation:**

****

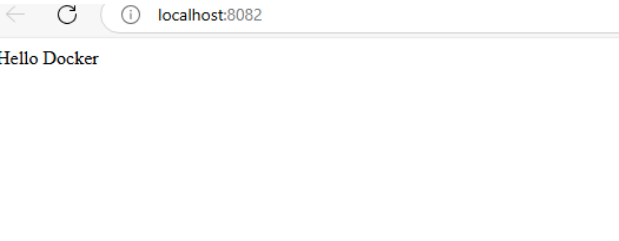
**Building Docker image**

****

**Running the container**

****

**Accessing rhe Apache Server that is run within docker container**

****

**Result:**

**Thus Simple HTML Page was containerized using docker**

|  |  |
| --- | --- |
| **Ex. No: 8** | **Applying CI/CD Principles to Web Development Using Jenkins, Git, and Local HTTP Server** |
| **dd.mon.year** |

**Aim : To Set up a CI/CD pipeline for a web development project using Jenkins, Git, and webhooks, without the need for a Jenkinsfile.**

**Procedure:**

* **Set Up the Web Application and Local HTTP Server using Apache or any other you comfortable with. Make sure it is configured right way, and its application is in running condition**
* **Set Up a Git Repository**
* **Install and Configure Jenkins**
* **Create a Jenkins Job using “Freestyle Project” type. Configure the job to use a webhook trigger. (Use "GitHub hook trigger for GITScm polling")**
* **Set Up the Jenkins Job (Using Execute Shell)**
  + **In the job configuration, go to the "Build" section.**
  + **Add a build step of type "Execute shell."**
  + **In the "Command" field, define the build and deployment steps using shell commands. For example:**

**# Checkout code from Git**

**# Build your web application (e.g., npm install, npm run build)**

**# Copy the build artefacts to the local HTTP server directory**

***rm -rf /var/www/html/webdirectory/\****

***cp -r \* /var/www/html/webdirectory/***

* **Set Up a Webhook in Git Repository**

1. **In your Git repository (e.g., on GitHub), go to "Settings" and then "Webhooks."**
2. **Create a new webhook, and configure it to send a payload to the Jenkins webhook URL (usually http://jenkins-server/github-webhook/). (Make sure to set the content type to "application/json.")**
3. **OR use “GitHub hook trigger for GITScm polling?” Under Build Trigger**

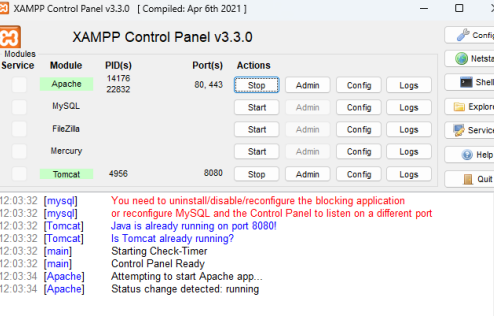
* **Trigger the CI/CD Pipeline**
  1. **Push changes to your Git repository.**
  2. **The webhook should trigger the Jenkins job automatically, executing the build and deployment steps defined in the "Execute Shell" build step.**
  3. **Monitor the Jenkins job's progress in the Jenkins web interface.**

* **Verify the CI/CD Pipeline**

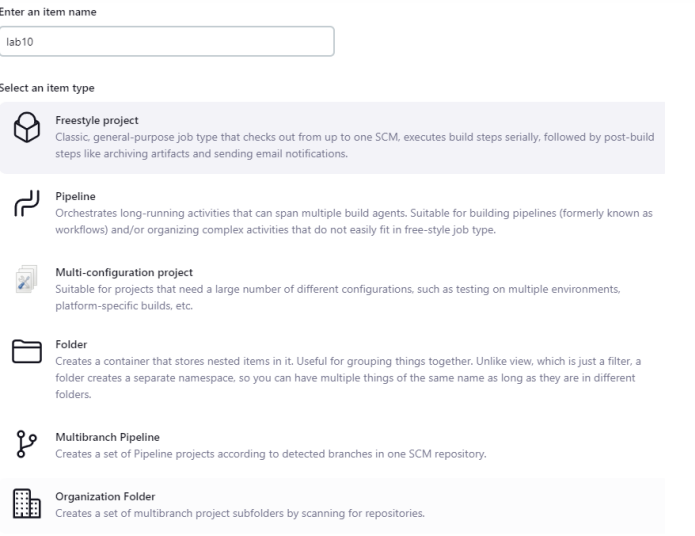
**Visit the URL of your local HTTP server to verify that the web application has been updated with the latest changes.**

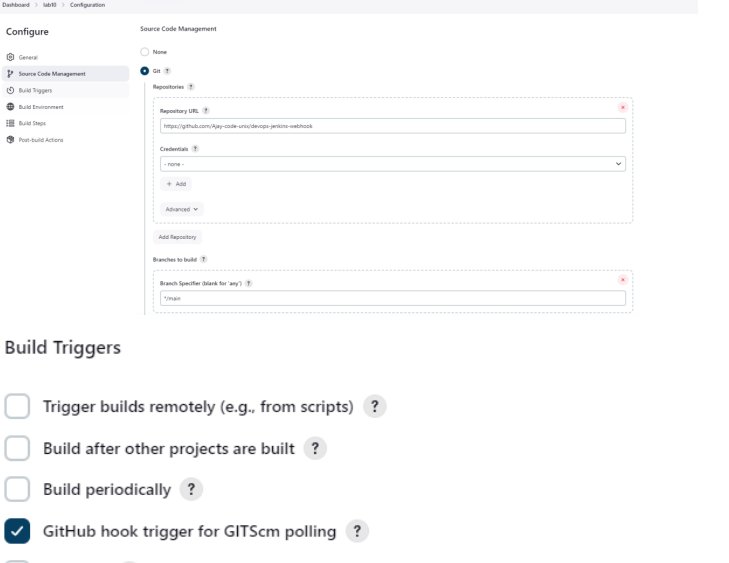
**Output:**

**Setting Up Local HTTP Server using xampp**

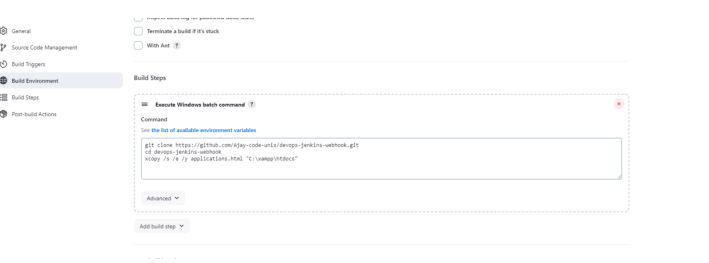
****

**Setting Up Git repository and Configuring Jenkins**

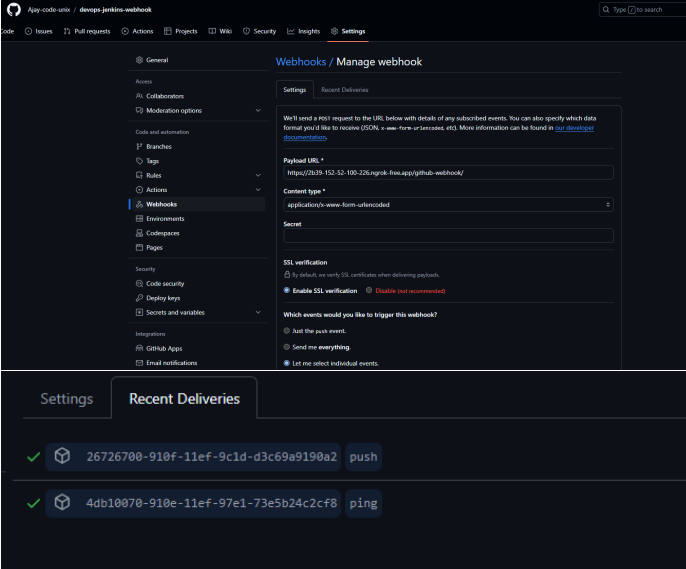
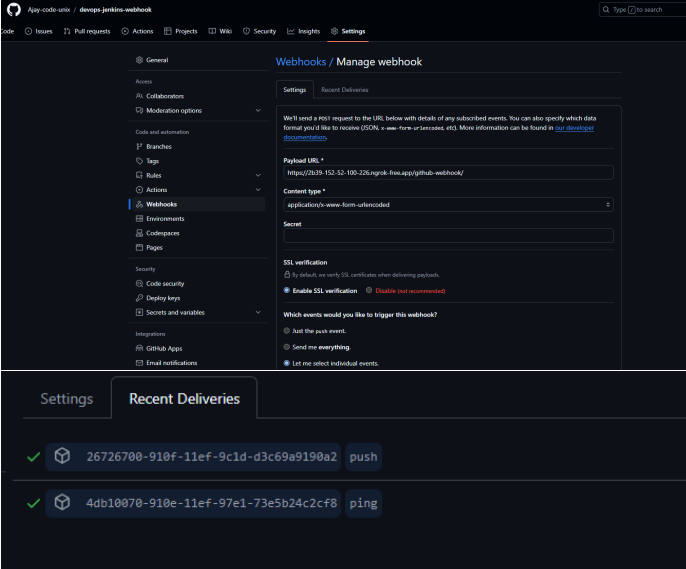
****

****

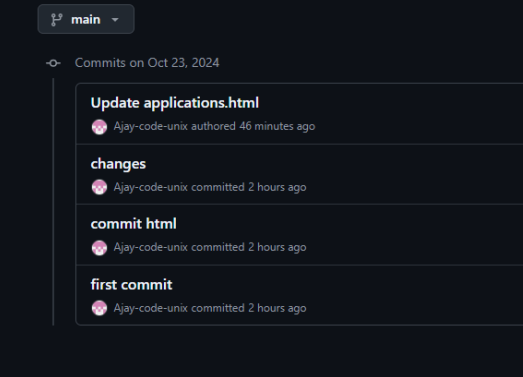
Adding Build Steps



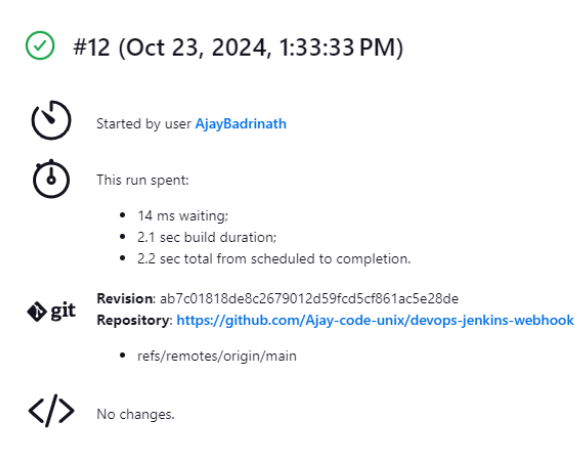
Setting up webhook in repository



Trigger the CI/CD Pipeline



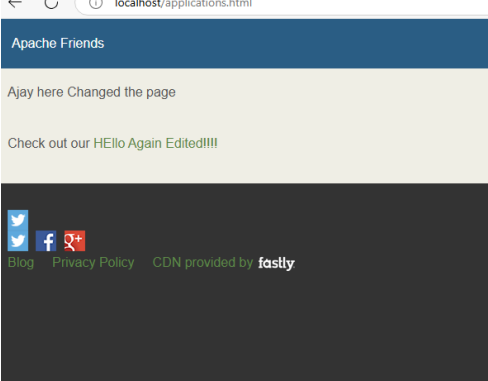
Jenkins job completed



Monitoring the build in the interface



Application has been updated with the latest changes



|  |  |
| --- | --- |
| **Ex. No: 5** | **Java Build Clean Lifecycle** |
| **dd.mon.year** |

Aim : To explore the build clean lifecycle of Maven Project

Algorithm

1.Create Java Project

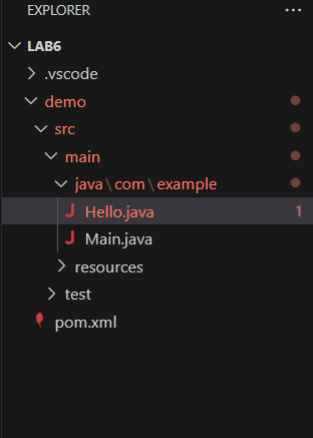
2.Create pom.xml add package as dependency

3.Check maven version

4.Execute Clean Lifecycle

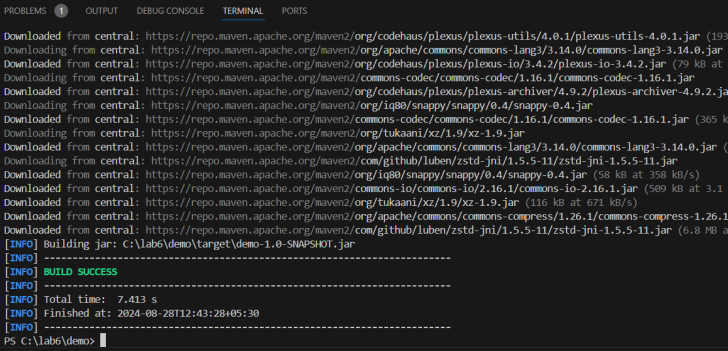
5.Explore maven clean Plugin

Output:

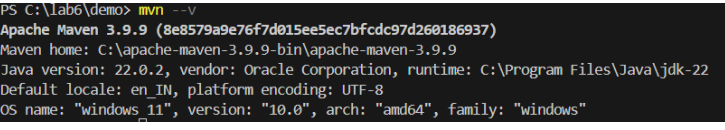


Pom file

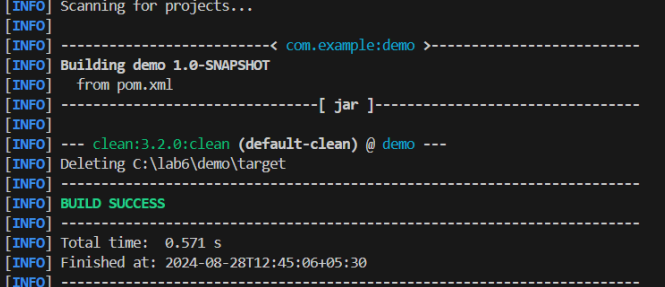




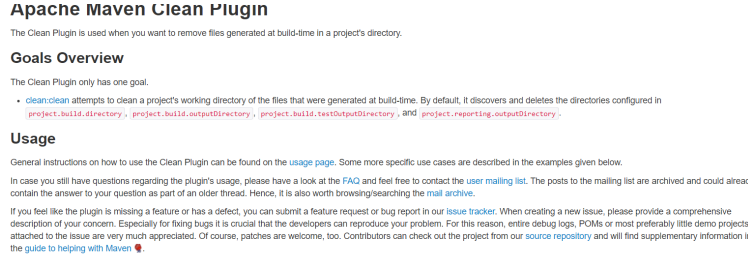
Check maven version



Executing clean lifecycle



Exploring Maven clean plugin



|  |  |
| --- | --- |
| **Ex. No: 4** | **Exploring Maven inheritance and Aggregation** |
| **dd.mon.year** |

Aim : To Explore maven Inheritance and Aggregation Properties

Procedure:

1.Define a Pom File with necessary artefact name group id

2. For Inheritance Create 2 other Child Pom in that import <Parent> tag referenceing the parent pom group id

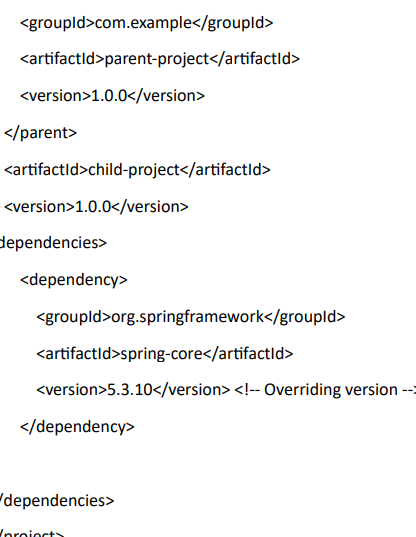
3. for Aggregation

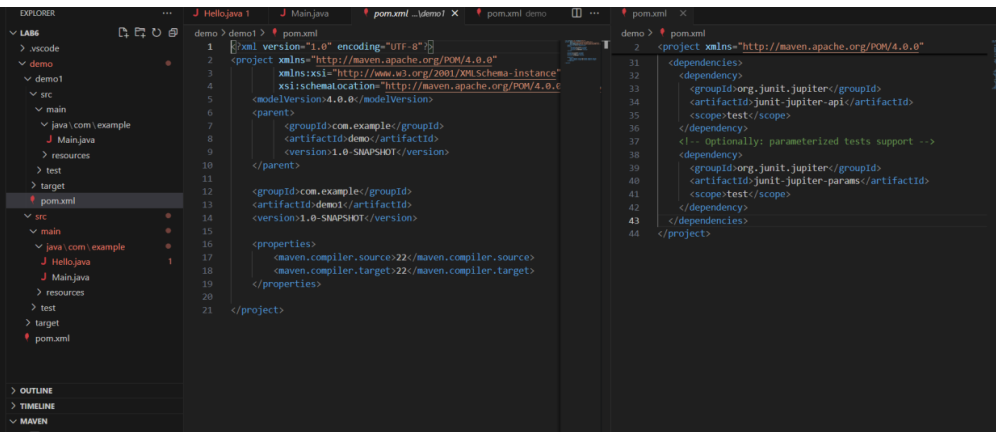
Output

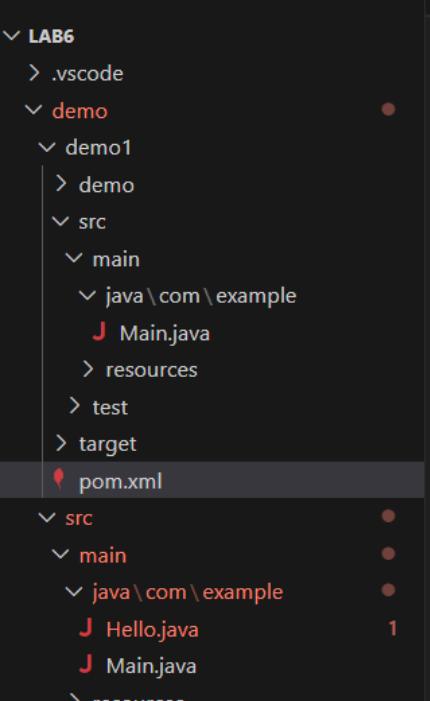
Parent pom



Child pom







Demo 1 is the child of Demo

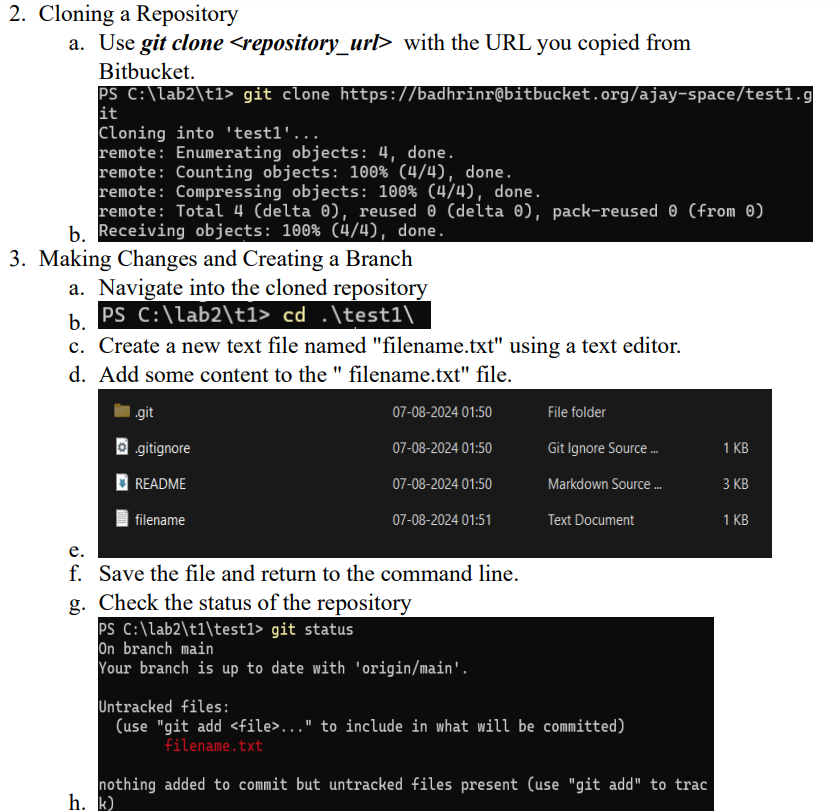
|  |  |
| --- | --- |
| **Ex. No: 3** | **Implement BitBucket Operations using Git** |
| **dd.mon.year** |

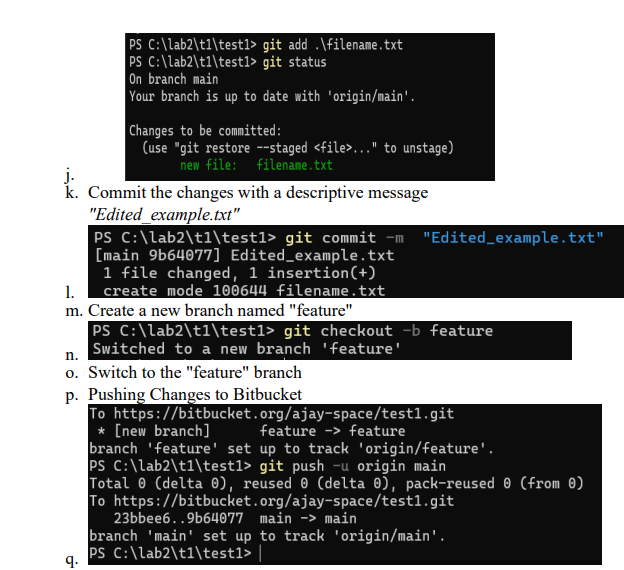
Aim : To implement Bit Bucket Operations using git

Procedure:

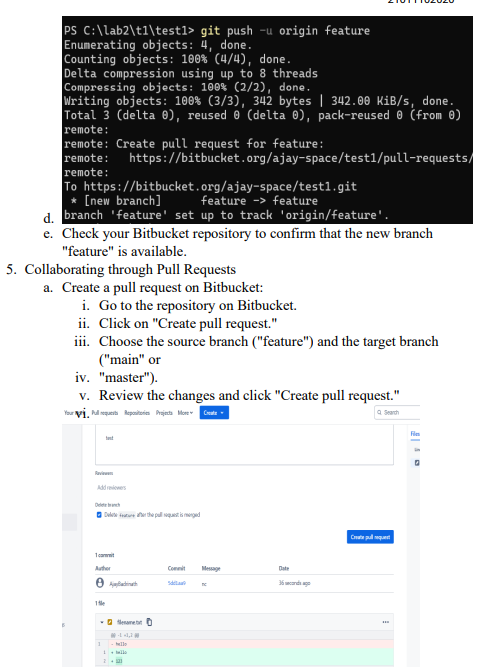
1. Creating a Repository
2. Cloning a Repository
   1. Use ***git clone <repository\_url>***  with the URL you copied from Bitbucket.
3. Making Changes and Creating a Branch
   1. Navigate into the cloned repository
   2. Create a new text file named "filename.txt" using a text editor.
   3. Add some content to the " filename.txt" file.
   4. Save the file and return to the command line.
   5. Check the status of the repository
   6. Stage the changes for commit
   7. Commit the changes with a descriptive message *"Edited\_example.txt"*
   8. Create a new branch named "feature"
   9. Switch to the "feature" branch
   10. Pushing Changes to Bitbucket
4. Pushing Changes to Bitbucket
   1. Add Repository URL in a variable
   2. Replace <repository\_url> with the URL you copied from Bitbucket
   3. Push the "feature" branch to Bitbucket
   4. Check your Bitbucket repository to confirm that the new branch "feature" is available.
5. Collaborating through Pull Requests
   1. Create a pull request on Bitbucket:
      1. Go to the repository on Bitbucket.
      2. Click on "Create pull request."
      3. Choose the source branch ("feature") and the target branch ("main" or
      4. "master").
      5. Review the changes and click "Create pull request."
   2. Review and merge the pull request:
6. Add a title and description for the pull request.
7. Assign reviewers if needed.
8. Once the pull request is approved, merge it into the target branch.
9. Syncing Changes
10. After the pull request is merged, update your local repository

Output

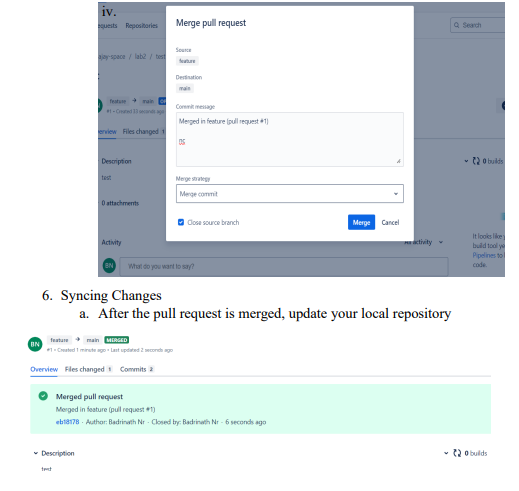




Pushing changes to Bit Bucket



Review and merge PR



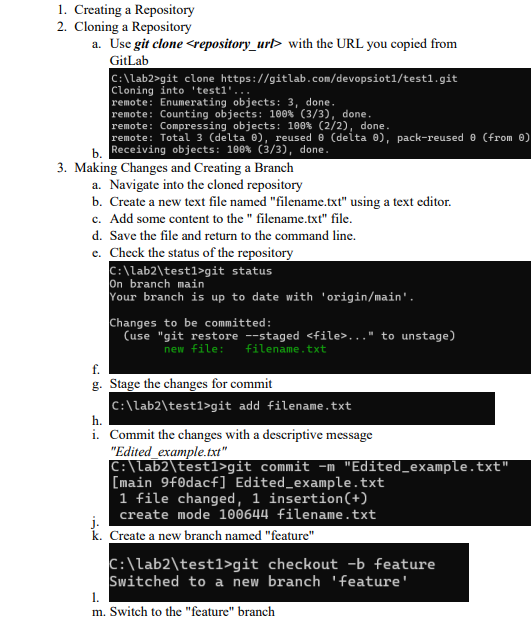
|  |  |
| --- | --- |
| **Ex. No: 2** | **Implement Git Lab Operations using Git** |
| **dd.mon.year** |

Aim : To implement Gitlab operations using Git SCM

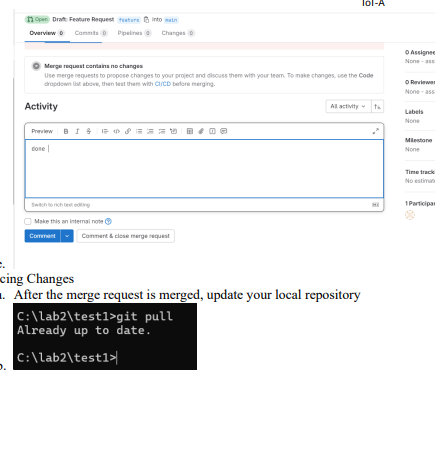
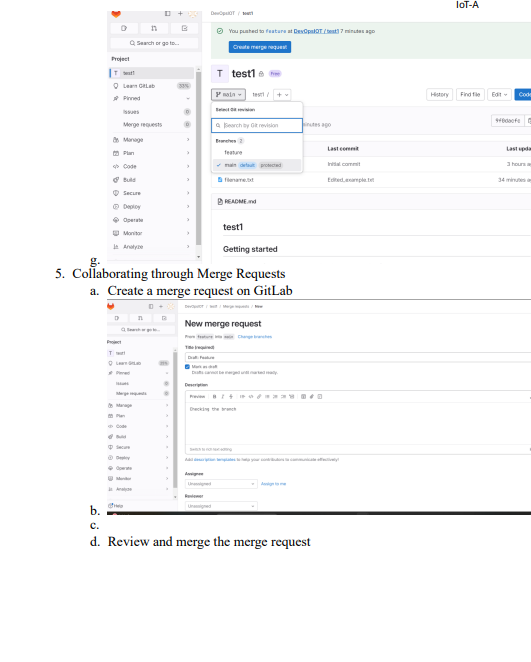
Procedure

1. Creating a Repository
2. Cloning a Repository
   1. Use ***git clone <repository\_url>***  with the URL you copied from GitLab
3. Making Changes and Creating a Branch
   1. Navigate into the cloned repository
   2. Create a new text file named "filename.txt" using a text editor.
   3. Add some content to the " filename.txt" file.
   4. Save the file and return to the command line.
   5. Check the status of the repository
   6. Stage the changes for commit
   7. Commit the changes with a descriptive message *"Edited\_example.txt"*
   8. Create a new branch named "feature"
   9. Switch to the "feature" branch
   10. Pushing Changes to GitLab
4. Add Repository URL in a variable
   1. Add Repository URL in a variable
   2. Replace <repository\_url> with the URL you copied from GitLab
   3. Push the "feature" branch to GitLab
   4. Check your GitLab repository to confirm that the new branch "feature" is available.
5. Collaborating through Merge Requests
   1. Create a merge request on GitLab
   2. Review and merge the merge request
6. Syncing Changes
   1. After the merge request is merged, update your local repository

Output:







|  |  |
| --- | --- |
| **Ex. No: 1** | **Implement Git Hub Operations using Git** |
| **dd.mon.year** |

Aim : To Implement Git Hub Operations using Git SCM

Procedure :

1. **Exploring Git Commands through Collaborative Coding – Basic Git commands.**
2. Create a Repository

* Setting Up Git Repository named “Master”

1. Add and Commit Changes

* Create a new text file named ReadMe.txt using any text editor.

1. Exploring history

* Modify/add content to the textfile.

1. Branching and merging

* Create a new branch named "Updated\_ReadMe" and commit the changes in the branch.
* Merge the changes from the Updated\_ReadMe branch into the Master branch.

1. Collaborating with Remote Repositories

* Link your local repository to the remote repository

1. Push Changes

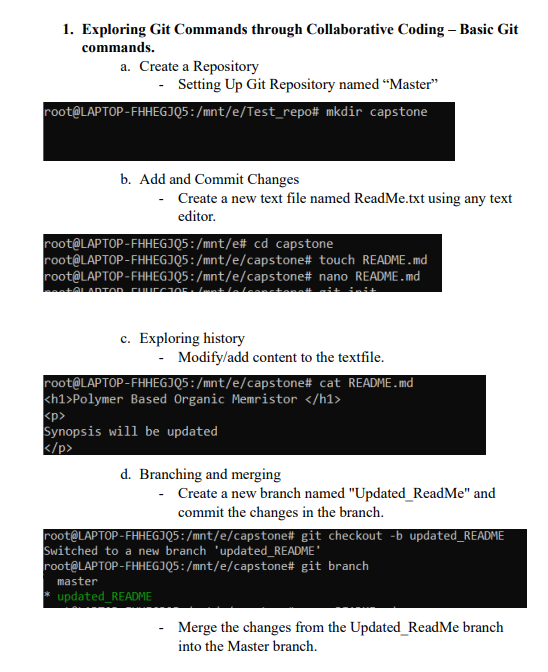
* Push your local commits to the remote repository

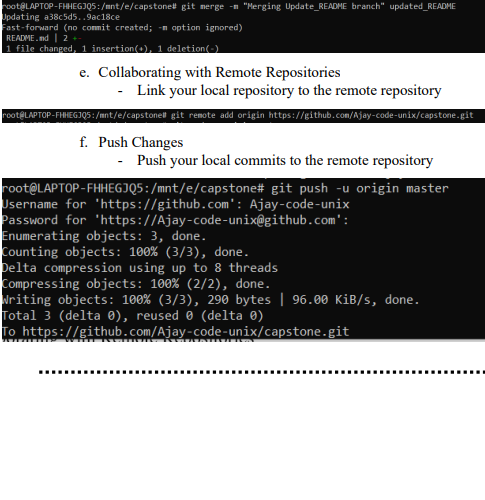
1. 2. **Implement GitHub Operations using Git.**
   1. **Cloning a Repository**

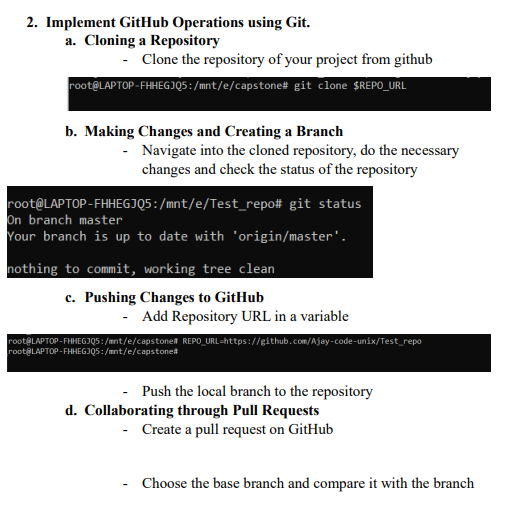
* Clone the repository of your project from github
  1. **Making Changes and Creating a Branch**
* Navigate into the cloned repository, do the necessary changes and check the status of the repository
  1. **Pushing Changes to GitHub**
* Add Repository URL in a variable
* Push the local branch to the repository
  1. **Collaborating through Pull Requests**
* Create a pull request on GitHub
* Choose the base branch and compare it with the branch
* Review the changes and merge the pull request.
* Add a title and description for the pull request
* Assign reviewers
* Reviewers approve the pull request, merge it into the base branch.
  1. **Syncing Changes**

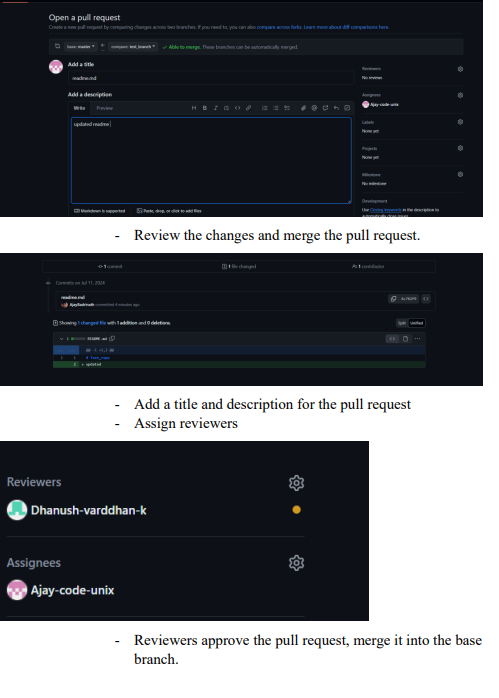
After the pull request is merged, update your local repository

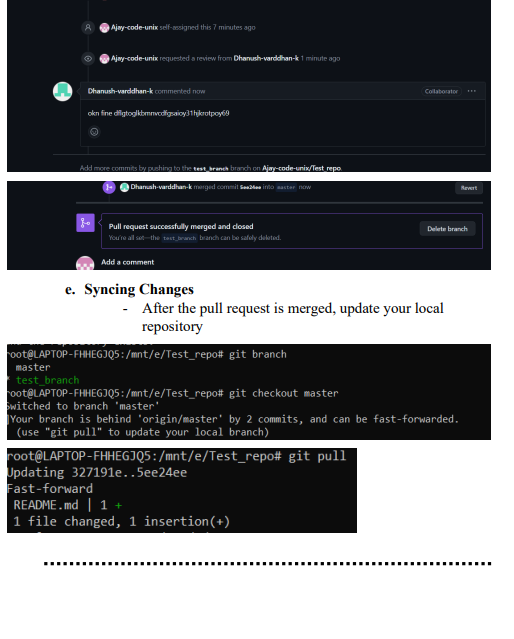
Output:











Result: Thus Github Operations were done using GIT SCM