**CBD-2244 DevOps Fundamentals for Canadian Enterprises**

**Node JS Application using CI / CD Pipeline**

**Professor Mohammed Salim**

**Group-1**

Shravya C0910891 (Team Leader)

Bhavani C0901954

Vakula C0911182

Deepak C0904124

**April 16, 2024**

**Abstract**

Continuous Integration/Continuous Deployment (CI/CD) automates different phases of the software development lifecycle to speed up the process. More than that, this project's primary goal is to build a pipeline for continuous integration and deployment for a Node.js application with post-deployment testing. Docker containers will be used to deliver the Node.js application. The application is hosted in the cloud using AWS EC2 instances, while Jenkins is used for continuous integration and delivery, and Docker is used for building docker images and containerization. Node JS libraries are used for testing.

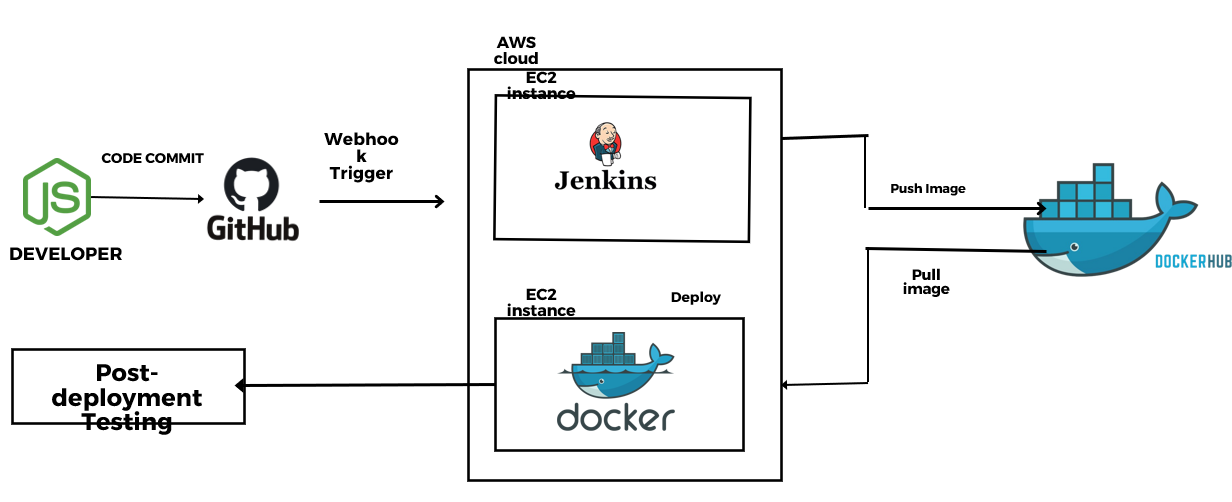
**Introduction**

Developing a Continuous Integration/Continuous Deployment (CI/CD) pipeline for the Node JS application is the primary goal of this project. This project aims to guarantee quality and stability while automating every step of the software development lifecycle, from modifying code to deployment. Building, testing, and deploying the application are only a few steps that make up the CI/CD process. For this project, we automate the process using cutting-edge tools and technologies like Docker, Jenkins, and Git.

The application's build and deployment procedures will be automated by the CI/CD pipeline, guaranteeing that the most recent updates are seamlessly merged, tested, and deployed. Jenkins will be the tool for continuous integration, starting builds, testing, and deployments whenever changes are added to the source repository. Jenkins automatically pulls updates from the version control system GIT, where the source code is committed.

Numerous project parts will benefit the clients, such as fewer human tasks, faster and more dependable deployments, and higher-quality applications. The efficiency and dependability of the application development and deployment process would, therefore, be significantly increased by building a CI/CD pipeline for Node JS application. Fault tolerance and high availability are essential features that enhance the end-user experience.

**CI / CD Pipeline Architecture**

****

The developer commits code to GitHub; the webhook triggers the Jenkins pipeline, and Jenkins creates the docker image and pushes it to the docker hub. Now, this docker image is again pulled and deployed in another EC2 server, and then finally, post-deployment testing is done.

**Methodology**

**Step 1: Create the application using Node JS**

The application is created using Node JS, and the code is stored in the version control system (GITHUB) repository. The source code is available in the below GitHub URL (mentioned below)

* The application is a web page that uses Node JS to show our project details.

GitHub URL: [git@github.com:DeepakChandMarthala/NodeJs-Project.git](about:blank)

Sample Source code:

A screenshot of a computer

Description automatically generated

Fig 1: Source Code

**Step 2: Create the Virtual machines in AWS**

Two EC2 instances are created in the AWS cloud platform, i.e.., Main-Project and Main-Project-3. The Main-Project-3 instance is used for automation purposes utilizing the Jenkins pipeline, and the Main-Project instance is used for application deployment into another instance.

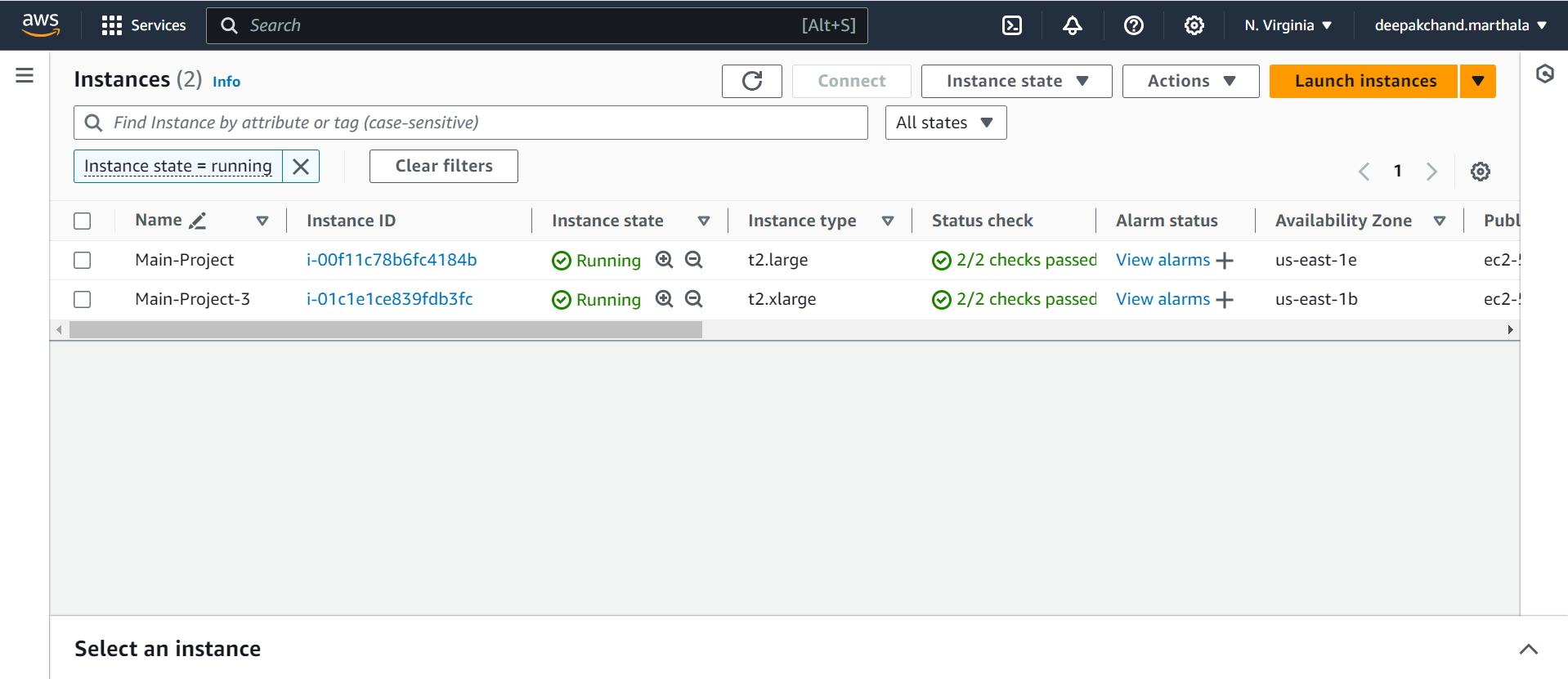


Fig 2: EC2 Instances in AWS

**Step 3: Pull the Source code from the GitHub repository to the Jenkins server.**

The developer develops the code and commits it to the GitHub repository. Jenkins is an open-source automation server for building, testing, and deploying software. It provides a wide range of plugins and integrations with other tools, allowing it to be customized to meet the needs. We used the Jenkins tool to automate the entire software development lifecycle, from building and testing code to deploying in Docker environments.

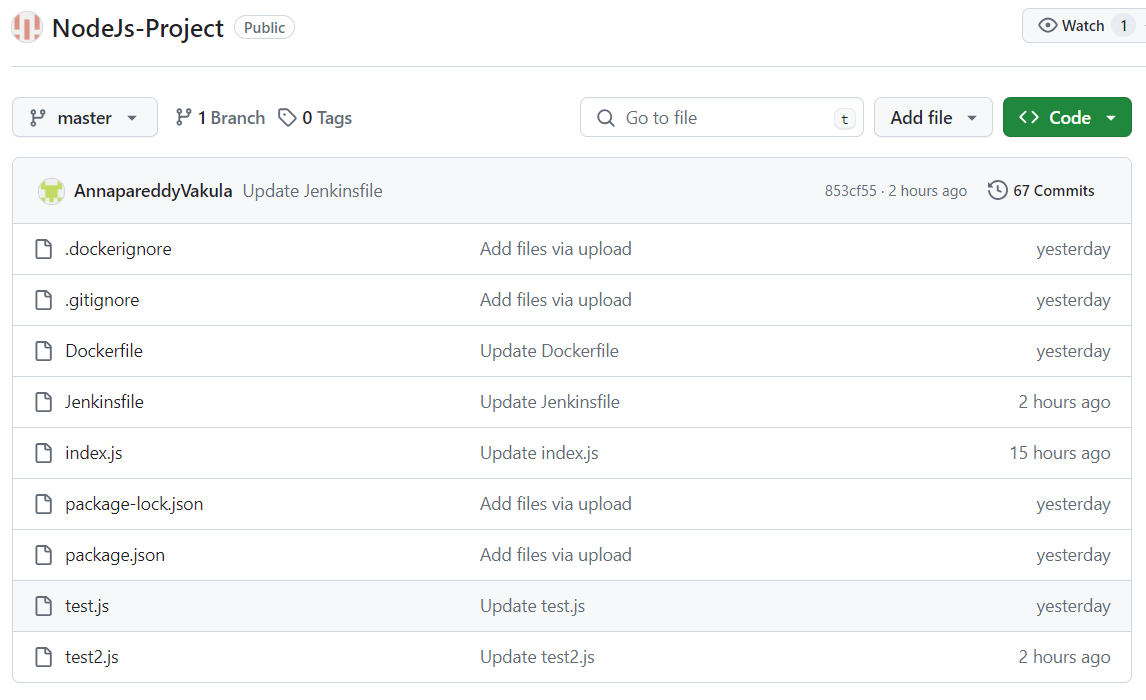


Fig 3: GitHub Repository

When changes are made to the source code stored in the GitHub repository, Jenkins will automatically pull the source code using the GitHub Webhook feature and initiate the Jenkins pipeline.

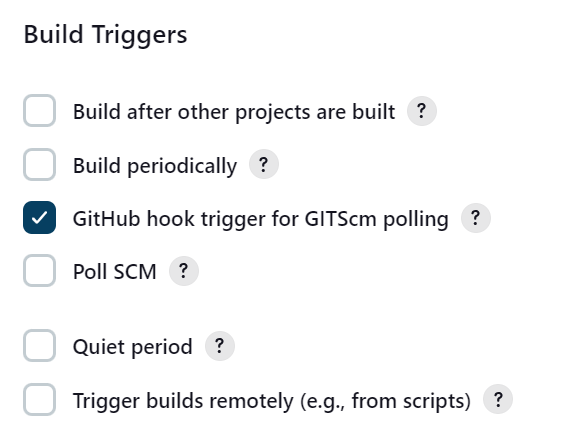


Fig 4: Webhook

**Stages in Jenkins:**

•  **Stage-1: Git Checkout SCM**

If any new changes are pushed to the GIT repository by the developer, Jenkins triggers the job using the webhook feature configured.

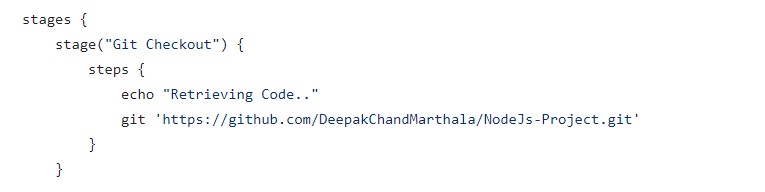


Fig 5: Retrieving the Github code

• **Stage-2: Build Docker Image**

Docker builds a new image based on the set of instructions provided in the docker file.

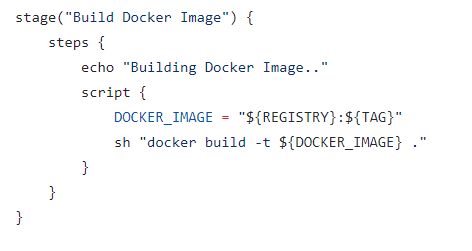


Fig 6: Building Docker Image

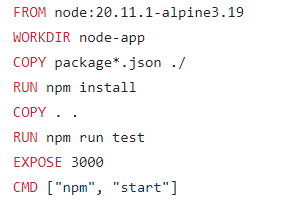
This code sets up a NodeJS environment on Alpine Linux, which provides lightweight. It installs dependencies, copies application files, and runs tests. The application listens on port 3000 and starts with the npm start command 

Fig 7: Docker File

The application is tested using the jest package using the GET method written in test.js. This code utilizes Jest and Super test to test an HTTP server, which imports Super test to simulate HTTP requests and Jest to write test cases. A test suite is defined to check the response of the root path to a GET request. After all tests have run, it attempts to close the server to clean up resources, handling scenarios where the server may not have a close() method.

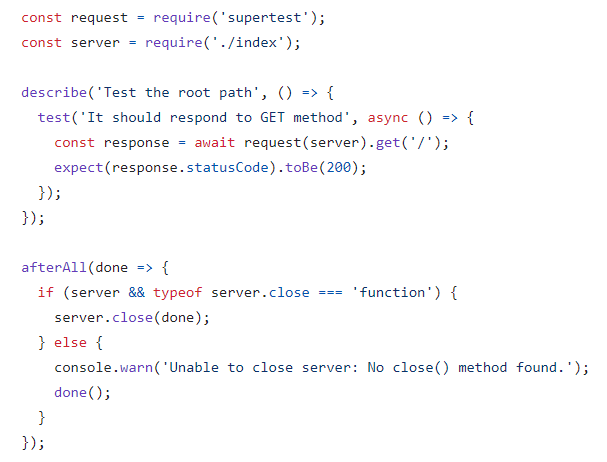


Fig 8: Test.js Code

Console output is provided below.

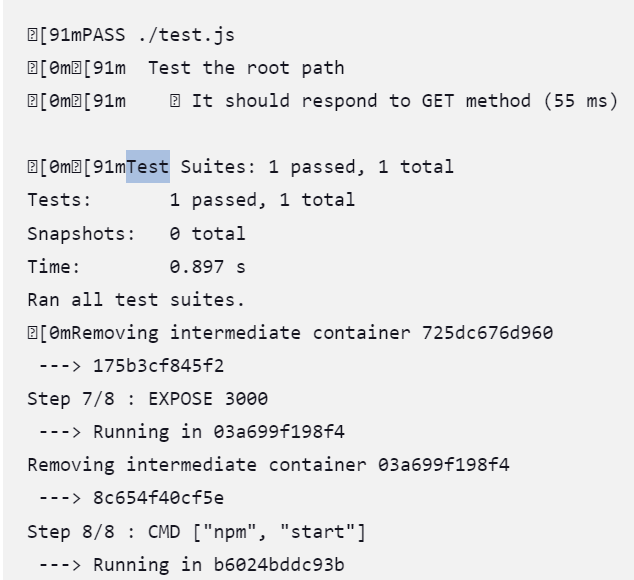
****

Fig 9: Console output of test.js in Jenkins pipeline

• **Stage-3: Login to Docker Hub**

Docker hub is being logged in using the credentials provided in the Jenkins Credentials, i.e.., Username and Access Token.

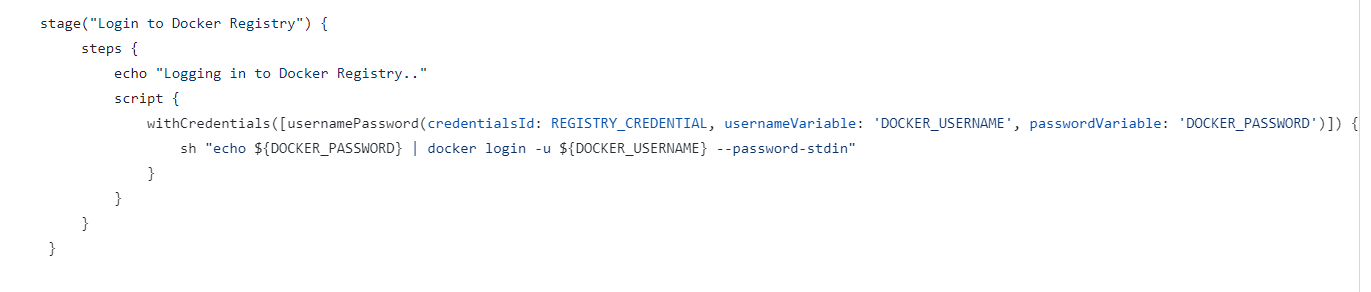


Fig 10: Logging into the docker registry

•  **Stage-4: Push the new Image to the Docker hub**

Docker will push the new image to Docker Hub

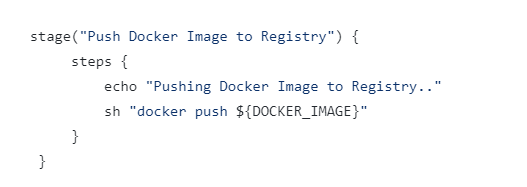


Fig 11: Pushing image into Docker Hub

The images that are stored in the docker hub with its tag name.

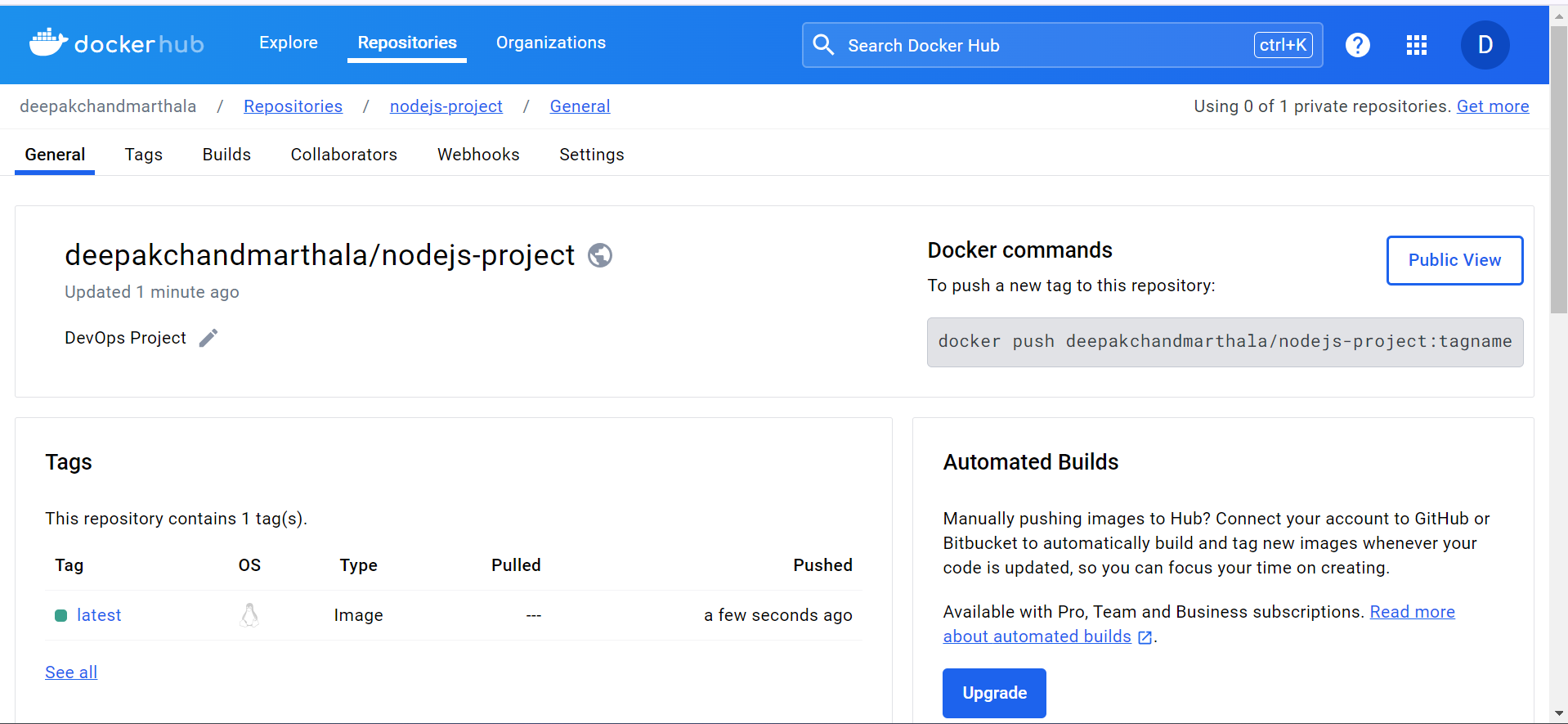


Fig 12: Docker Hub

• **Stage-5:** **Deploying in EC2 instance**



Fig 13: Deployment Code

Developers may quickly construct, deploy, and operate applications in containers with the help of the Docker software platform. Applications and their dependencies can run in lightweight, portable, self-contained environments called containers. Docker receives the updated source code from the Jenkins server.

The docker image present in the docker hub is used to deploy the application in the EC2 instance, and updated changes are reflected in the application.

• **Stage-6: Testing**

The deployed application is tested using Axios and assert packages. Axios is a popular JavaScript library used for making HTTP requests from the browser of the node.js environment. In Node JS, the assert module is a built-in module that provides a set of assertion functions for testing and debugging purposes. This function helps validate the text using predicted output text and the behavior of the code.

The code snippet which is used to test the node JS application is:

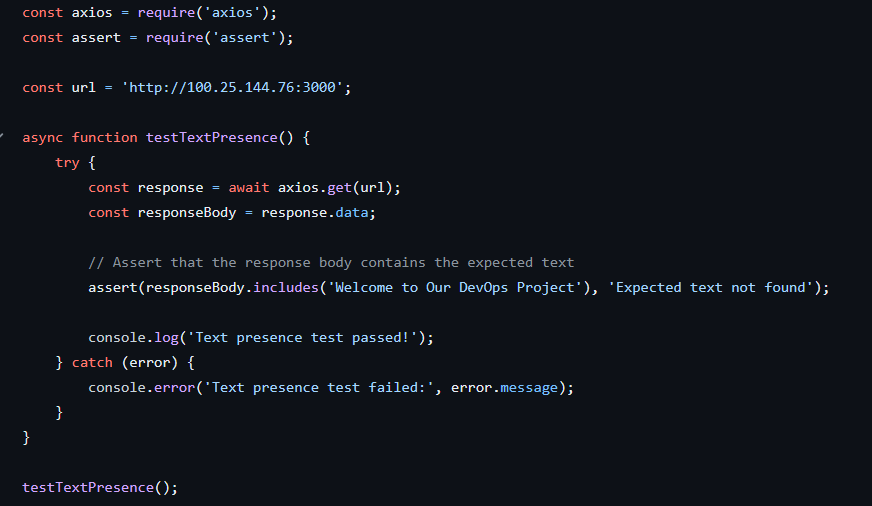


Fig 14: Post Deployment Testimg Code

The result console output that post-deployment testing is shown below:

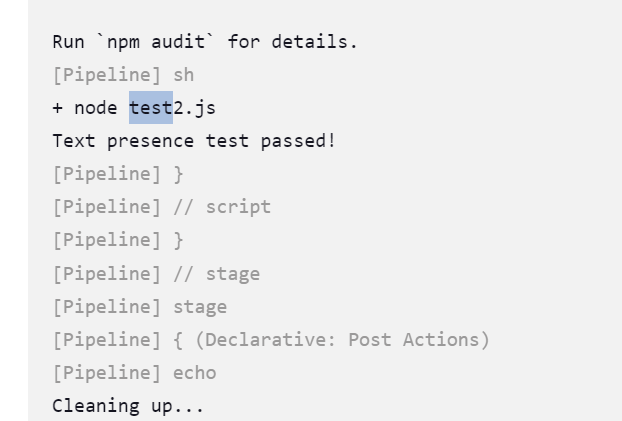


Fig 15: Console output of Deployment Testing

After the pipeline stages are done, the "docker logout" command logs out the current user from the Docker registry to ensure secure access. "docker system prune -a -f" removes all stopped containers, unused networks, and dangling images, freeing up disk space and cleaning the Docker environment.

Here are the final console output and Jenkins stages

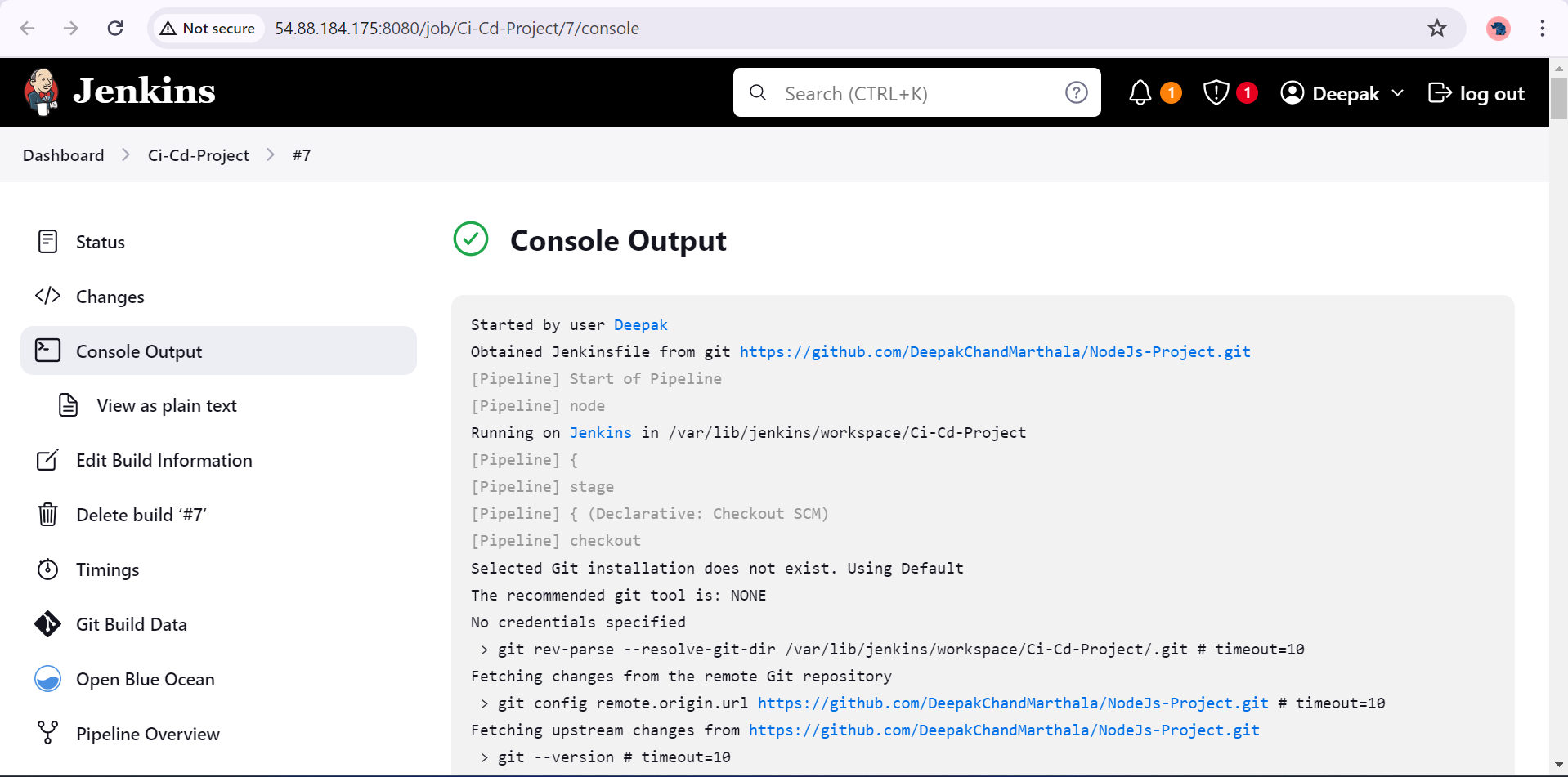


Fig 16: Console Output when Jenkins is started

All of the pipeline stages that are present in our pipeline to ensure the project follows continuous integration and Deployment

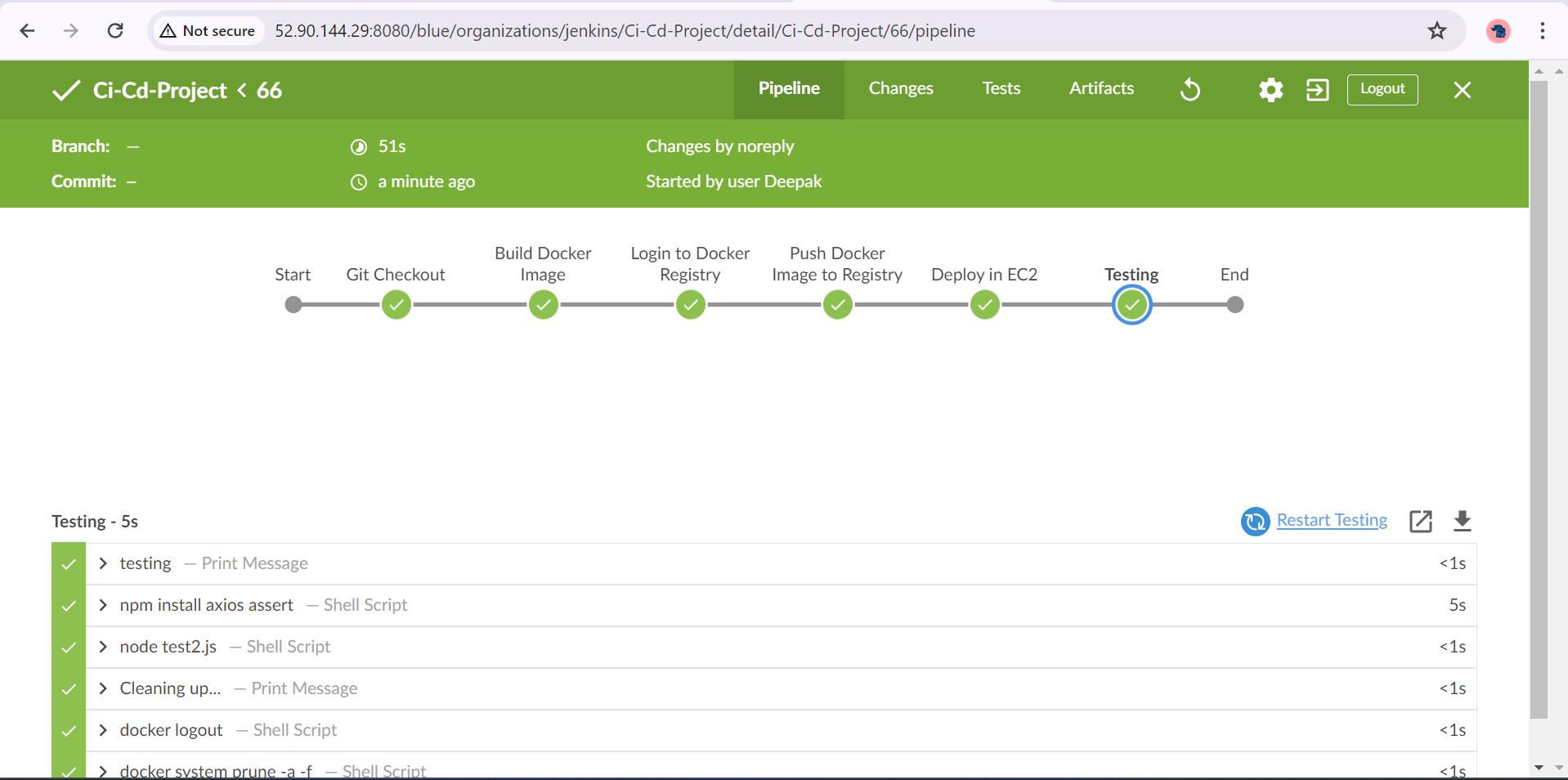


Fig 17: Jenkins Pipeline Stages

Deployed Application Output is hosted on the Amazon EC2 instance that is used for deploying the application with the port specified, i.e..., 3000.

A screenshot of a computer

Description automatically generated

Fig 18: Application Output

**Conclusion**

In conclusion, the Continuous Integration/Continuous Deployment (CI/CD) methodology is a crucial component of modern software development that allows process optimization by automating several phases of the software development lifecycle. By utilizing Docker containers and Node JS to create a pipeline for a Node JS application, this project illustrates how CI/CD can be used in real-world scenarios. The project uses various DevOps tools, including Jenkins, AWS, and Docker, to facilitate continuous integration and delivery. It is easy to construct the necessary infrastructure in AWS, such as virtual networks, subnets, security groups, and virtual machines. The software development process may be made simpler and software delivery more reliable and efficient using DevOps tools, as demonstrated by this project. By ensuring post deployment testing is done so that there is no mismatch between what is developed and what is deployed.

**Reason for Modification from Proposal**

1. **Source Code (From Java to Node JS)**

The source code application has been changed from Java to Node JS, as the command line output can be accessed only by logs or by opening the output file after deploying to EC2. If we need to make a user interface Java application that needs configuration changes, i.e.., installing dependencies, using an external server, and utilizing packages in Java where the same process in Node JS with few modules and its lightweight nature and rich ecosystem of libraries (such as npm) allow developers to iterate quickly and build features faster.

1. **Pre-Deployment and Post-Deployment Testing:**

Pre-deployment testing (using jest) ensures that the application meets quality standards, functionality, and performance benchmarks before deployment, reducing the risk of issues in production. Post-deployment testing has been performed using Axios and assert modules, which validate the deployed application's functionality in the live environment, identifying any defects that may have occurred during deployment. Together, they provide comprehensive validation of the application's readiness for production, ensuring a smooth and successful deployment process while maintaining the integrity and reliability of the software system.

1. **SonarQube**

Due to some configuration issues and time constraints, sonarqube is not included. This can be added to our project in the future along with other components. Being new to all the new tools and technologies, which took most of the time configuring.

**References**

* [**https://www.cprime.com/resources/blog/how-to-integrate-jenkins-github/**](https://www.cprime.com/resources/blog/how-to-integrate-jenkins-github/)
* [**https://docs.docker.com/docker-hub/**](https://docs.docker.com/docker-hub/)
* [**https://axios-http.com/docs/**](https://axios-http.com/docs/)
* [**https://www.jenkins.io/doc/**](https://www.jenkins.io/doc/)

**Appendix**

**Source code link:****https://github.com/DeepakChandMarthala/NodeJs-Project/blob/master/index.js**

**Docker hub repository:**[**https://hub.docker.com/repository/docker/deepakchandmarthala/nodejs-project/**](https://hub.docker.com/repository/docker/deepakchandmarthala/nodejs-project/)

**Source Code:**

const express = require('express');

const app = express();

const port = 3000;

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

const htmlContent = `

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>DevOps Project</title>

<style>

body {

font-family: Arial, sans-serif;

background-color: #f4f4f9;

margin: 40px;

color: #333;

}

h1 {

color: #5a5a5a;

}

.content {

background-color: #ffffff;

padding: 20px;

border-radius: 8px;

box-shadow: 0 4px 8px rgba(0,0,0,0.1);

}

p {

line-height: 1.6;

}

</style>

</head>

<body>

<div class="content">

<h1>Welcome to Our DevOps Project</h1>

<p>This project is done as a group by:</p>

<ul>

<li>Bhavani</li>

<li>Shravya</li>

<li>Vakula</li>

<li>Deepak Marthala</li>

</ul>

</div>

</body>

</html>

`;

res.send(htmlContent);

});

// Health check route

app.get("/health", (\_req, res) => {

res.send("Everything's good!");

});

// Start the server

const server = app.listen(port, () => {

console.log(`Example app listening on port ${port}`);

});

module.exports = server;