	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering (PCCoE)** (An Autonomous Institute) Affiliated to Savitribai Phule Pune University(SPPU) ISO 21001:2018 Certified by TUV	

Course: DevOps Laboratory	**Code:** BIT26VS01
Name: Amar Vaijinath Chavan	**PRN:** 124B2F001
Assignment 7: Study and Implementation of Declarative and Scripted Pipelines in Jenkins.	

Aim: To design a complete CI/CD lifecycle using a Declarative Jenkinsfile, including stages for automated Building, Testing, and Deployment of a Spring Boot application integrated with a MongoDB database.

Objectives:

1. To compare and contrast **Declarative** and **Scripted** Jenkins Pipelines.
2. To implement "Pipeline as Code" using a Jenkinsfile in a Master-Worker architecture.
3. To automate the deployment of a Spring Boot backend and verify data persistence in a MongoDB instance.

Theory:

1. Declarative vs. Scripted Pipeline Paradigms Jenkins offers two distinct syntaxes for defining pipelines: Declarative and Scripted. Declarative Pipelines are a more recent addition, providing a simplified, opinionated, and structured syntax using pipeline {} blocks, which makes them easier to read and maintain for beginners. In contrast, Scripted Pipelines use a more traditional Groovy-based syntax within node {} blocks, offering maximum flexibility and power for complex logic but requiring a higher degree of programming knowledge. For modern DevOps practices, Declarative Pipelines are generally preferred as they enforce a cleaner structure and provide built-in features for error handling and logging.

2. The "Pipeline as Code" Concept The core of modern Jenkins automation is the Jenkinsfile, which allows the entire build, test, and deployment process to be defined as code and stored directly within the source control repository (SCM). This practice, known as **Pipeline as Code**, ensures that the delivery pipeline is version-controlled, auditable, and easily reproducible. By keeping the Jenkinsfile in the root directory, the pipeline evolves alongside the application code, allowing developers to modify build stages through simple commits rather than manual UI adjustments.

3. **Master-Worker (Controller-Agent) Execution** In a distributed Jenkins environment, the Master (Controller) node manages the orchestration, UI, and job scheduling, while the **Worker (Agent)** nodes handle the actual execution of tasks. This separation is crucial for resource-intensive Java projects, as the worker node provides a clean, isolated environment for Maven compilation and testing. By targeting a specific agent—such as Jenkins-Worker-01—the pipeline ensures that the Master remains responsive while the heavy lifting of the CI/CD lifecycle is performed by scalable cloud instances.

4. **Automated Build Lifecycle with Maven** Maven serves as the primary build automation tool for Java applications, managing dependencies and project structure through the pom.xml file. Within the pipeline, the mvn clean package command initiates a series of phases: first, it clears previous build artifacts; then, it compiles the source code; and finally, it packages the application into a JAR file. This automation ensures that the code is always buildable and results in a standardized, deployable artifact that is ready for the testing and deployment stages.

5. **Continuous Testing and API Validation** Testing is a critical gate in any CI/CD pipeline, ensuring that new code changes do not break existing functionality. In this assignment, testing is performed from an "end-user perspective" using tools like Postman to validate REST API endpoints. By sending HTTP requests (POST/GET) to the running Spring Boot application, developers can verify that the backend logic is correct and that the application handles data as expected before it is officially marked as a successful deployment.

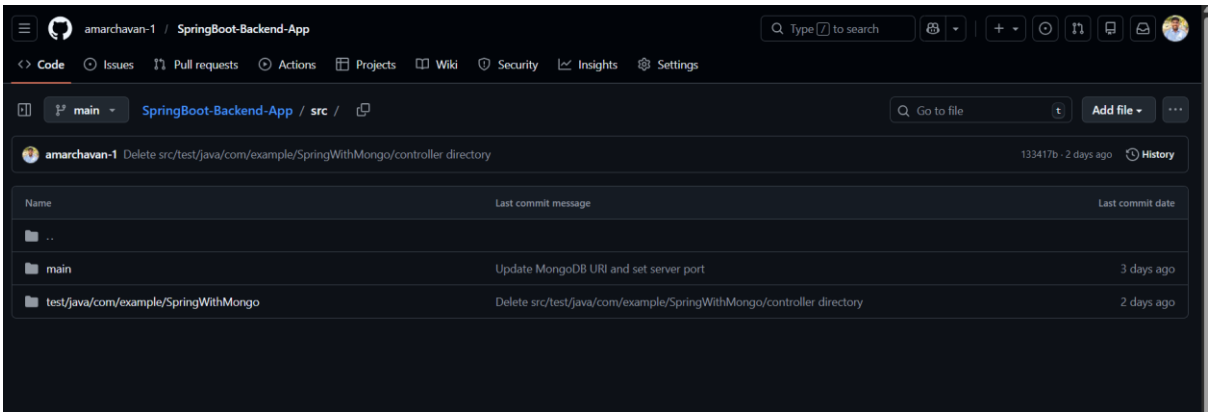
6. **Database Persistence with MongoDB** For full-stack applications, the deployment stage must include integration with a database to verify data persistence. **MongoDB**, a NoSQL database, is used here to store application data such as product catalogs. Verifying the database state using the mongosh utility on the EC2 instance ensures that the application is not only running but also successfully communicating with its data layer. This end-to-end verification confirms that the entire stack—backend, database, and infrastructure—is functioning correctly.

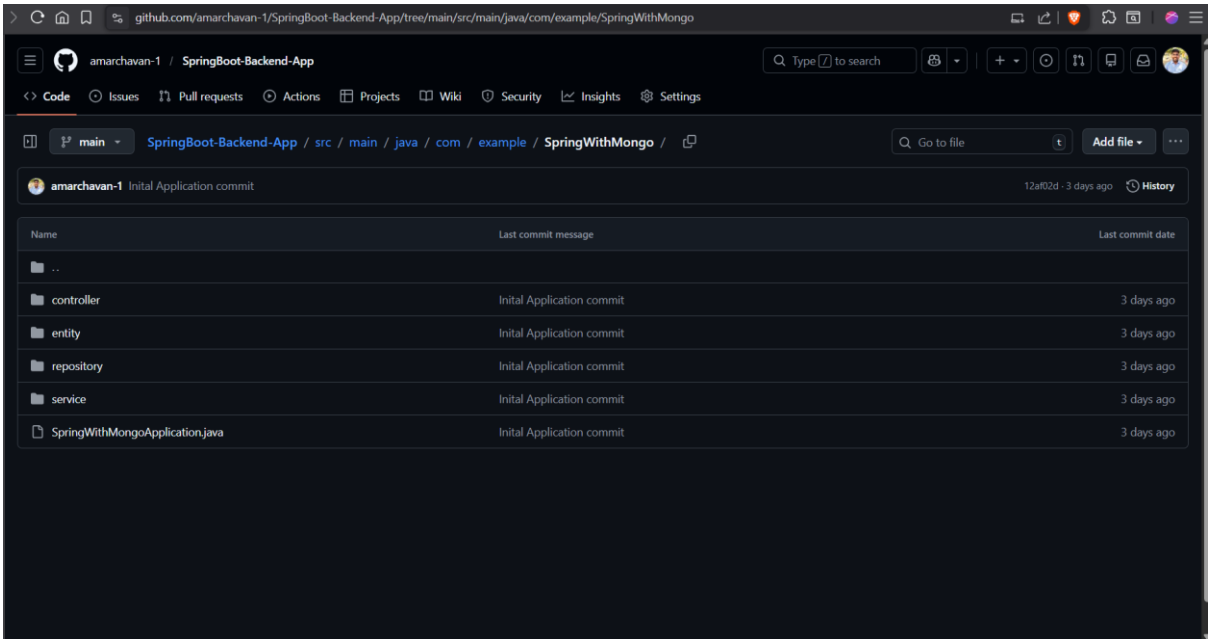
7. **Automated Triggers and Webhooks** The ultimate goal of a CI/CD pipeline is to remove manual intervention through automated triggers. By configuring **GitHub Webhooks**, a "push" event in the repository automatically signals Jenkins to initiate the pipeline. This real-time synchronization allows for immediate feedback; as soon as a developer commits code, the pipeline pulls the latest changes, builds the project, and deploys it, ensuring that the most recent version of the software is always available for testing and review.

Practical Procedure / Steps:

Step 1: GitHub & SCM Configuration

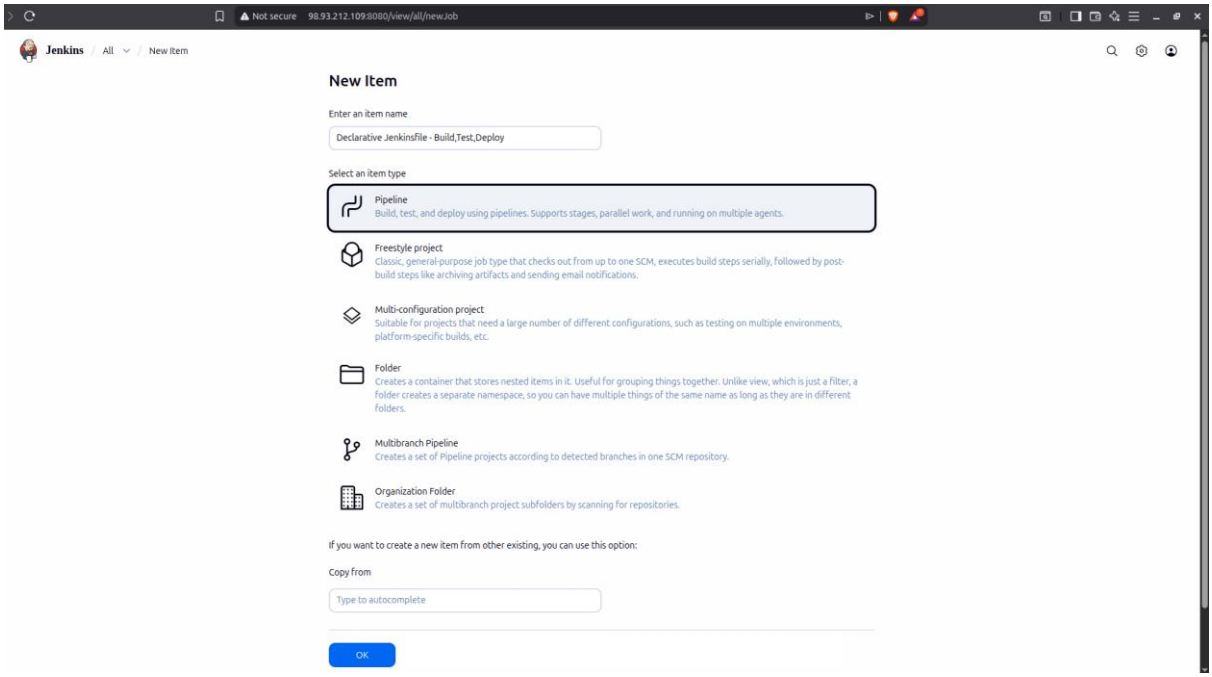
- Ensure the **Personal Access Token (PAT)** is configured in Jenkins under github-creds.
- ***Note***:- *Proper Working Backend/Frontend/or fullstack application is required for performing this build-test-deploy assignment with the test cases in the application [if the test cases are not in the application it will work fine but the test cases are not run]*





Step 2: Jenkins Pipeline Creation

1. Create a new **Pipeline** item named Declarative-Jenkinsfile-Build-Test-Deploy.



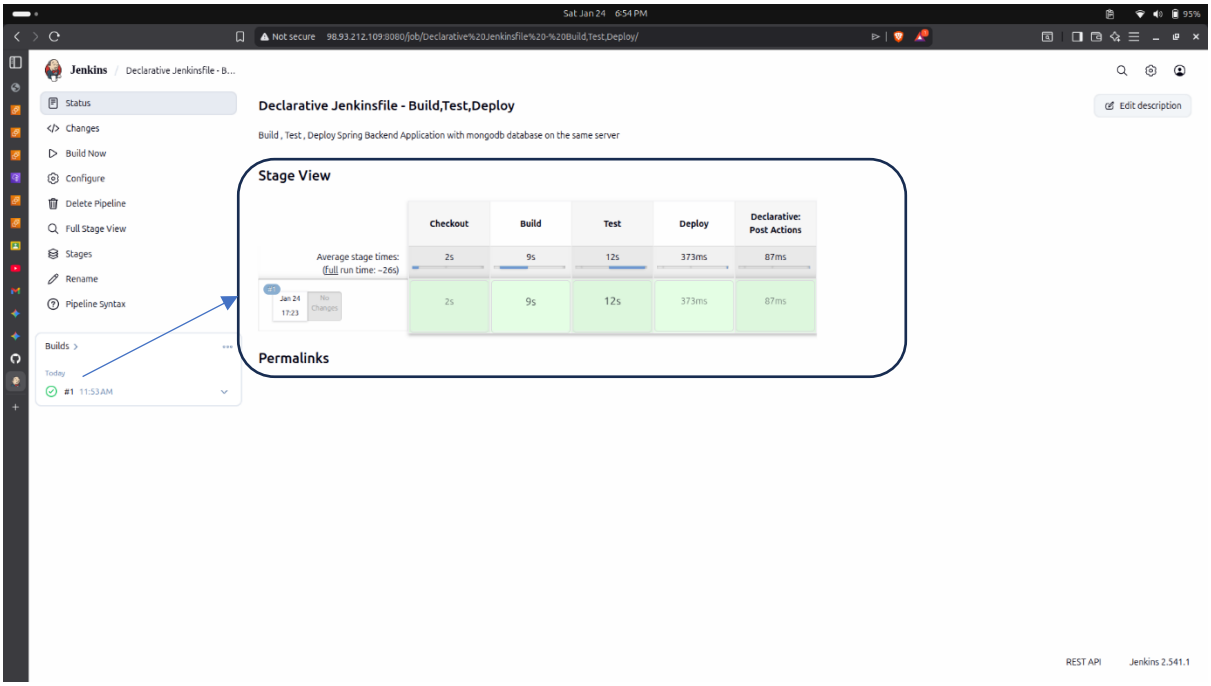
Add This Pipeline Code:

```
pipeline {
    agent { label 'Jenkins-Worker-01' }

    triggers {
        githubPush()
    }

    stages {
        stage('Checkout') {
            steps {
                    echo 'Pulling latest code from GitHub...'
                    git branch: 'main',
                    credentialsId: 'github-creds',
                    url: 'https://github.com/amarchavan-1/SpringBoot-Backend-App.git'
            }
        }

        stage('Build') {
            steps {
                echo 'Compiling the application...'
                sh 'mvn clean compile'
            }
        }

        stage('Test') {
            steps {
                echo 'Executing JUnit and Integration Tests...'
                sh 'mvn test'
            }
        }
```

```
        stage('Deploy') {
            steps {
                echo 'Deploying to Worker Node on Port 8081...'
                sh '''
                    fuser -k 8081/tcp || true
                    nohup java -jar target/*.jar > app.log 2>&1 &
                '''
            }
        }
    }

    post {
        success {
            echo 'Assignment 7: All Tests Passed and App Deployed!'
        }
        failure {
            echo 'Pipeline Failed: Tests failed or Build error detected.'
        }
    }
}
```

Step 3: Execution and Monitoring

1. Trigger the build manually or via a code commit to GitHub.
2. Monitor the **Stage View** in the Jenkins UI to track progress through Checkout, Build, Test, and Deploy stages.

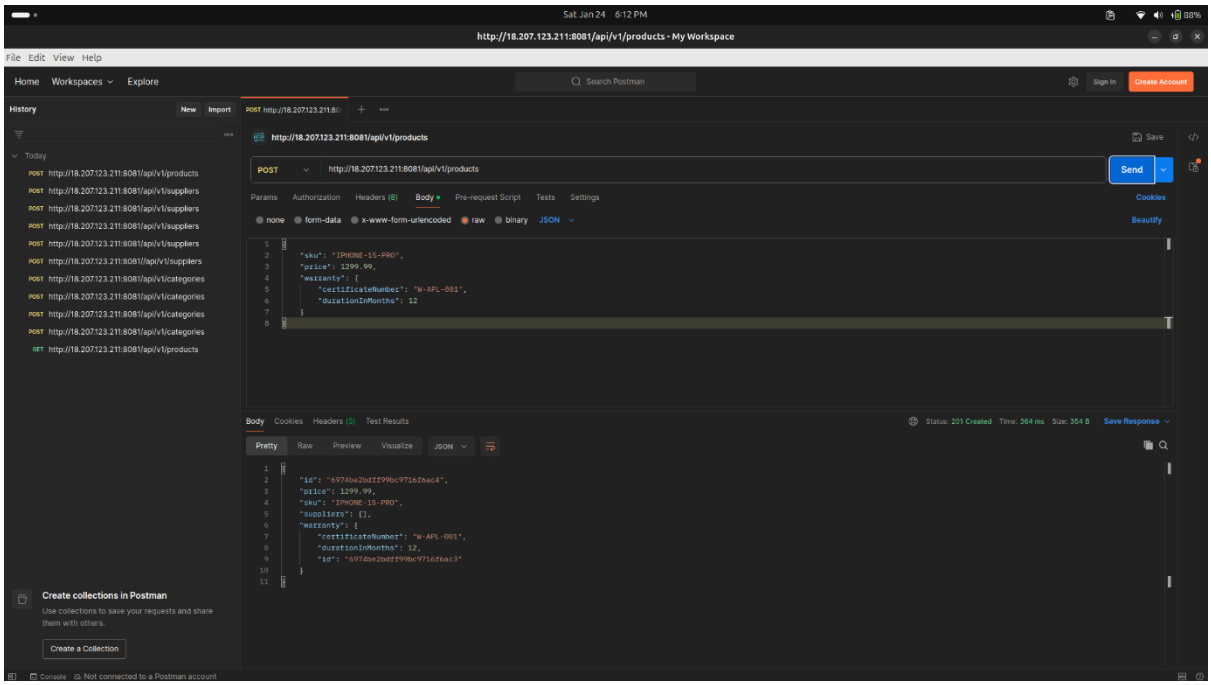


Step 4: API Validation and Server Verification

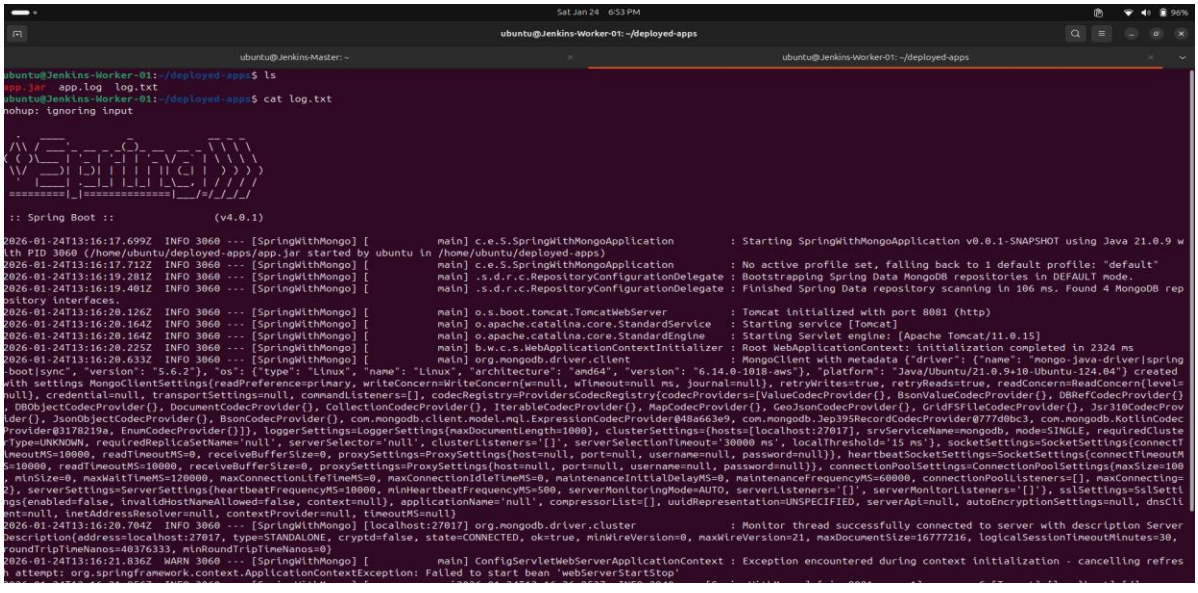
Note:- *Application Overview:*

The application used in this assignment is a Spring Boot CRUD (Create, Read, Update, Delete) Backend. It serves as a product catalog management system where users can perform operations on product data via RESTful APIs. The application is integrated with MongoDB for data persistence and is designed to run on a distributed Jenkins Master-Worker architecture.

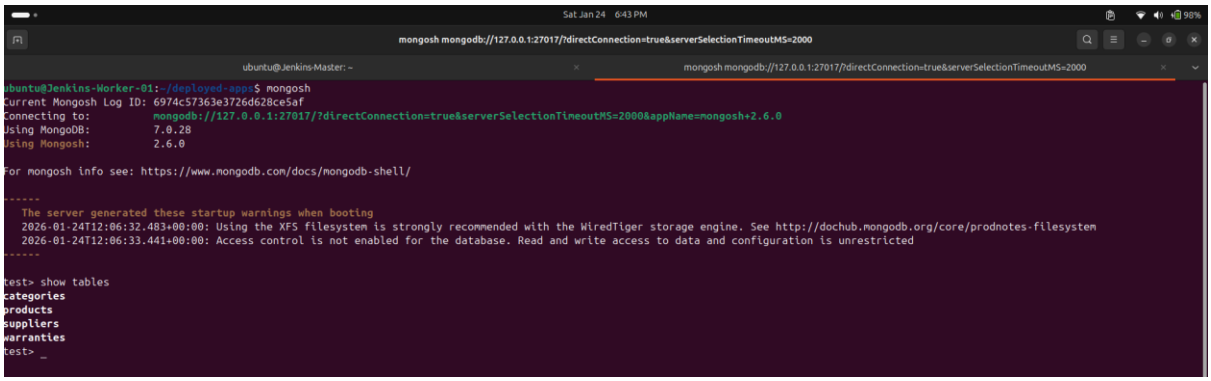
1. **API Testing (Postman):** Use Postman to send a **POST** request to http://<Worker-IP>:8081/api/v1/products with a JSON payload. Verify the **201 Created** response.

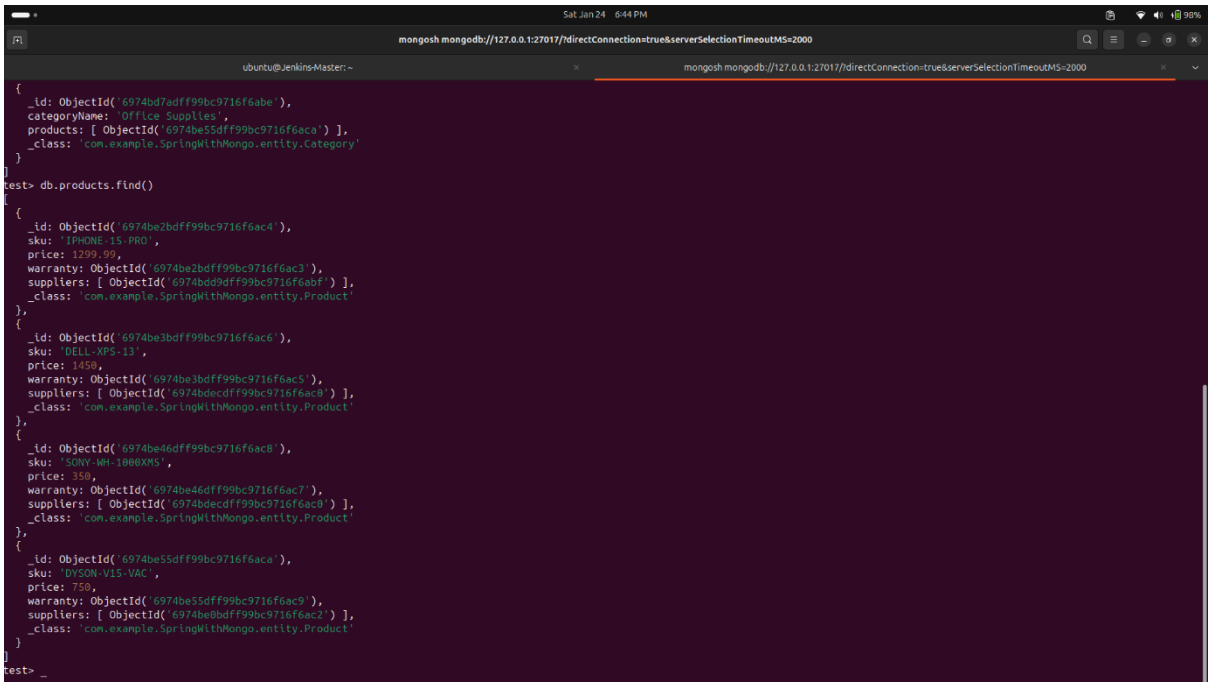


2. **Server Logs:** Access the worker node terminal and run cat app.log. Verify the Spring Boot banner and the message: Tomcat initialized with port 8081 (http).



3. **Database Verification:** Use the mongosh utility on the server and run db.products.find(). Confirm that the data sent via Postman is physically stored in the MongoDB collection.





All API's Test Screenshots:-
https://drive.google.com/drive/folders/1acxnnqLZ0NsCCQKZJGNbN5WNu1lKmQA3?usp=sharing

Conclusion

The execution of this assignment successfully demonstrates a full-scale **Continuous Deployment** pipeline using a **Declarative Jenkinsfile** to manage a Spring Boot and MongoDB stack. By adopting the **Pipeline as Code** model, we established a structured workflow where stages for building, testing, and deploying are defined in a central script and executed through the Jenkins UI. Offloading these resource-intensive tasks to a dedicated **Worker Node** ensured that the Jenkins Master remained optimized for orchestration while the application was compiled and tested in an isolated environment. Verification through **Postman** and **MongoDB** inspection proved that the application successfully maintained API functionality and data persistence throughout the automated deployment process.