	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 6: Build Job using Jenkins that compiles a Java project and configure triggers to run the build on every commit.	

Aim: To configure a Jenkins Pipeline that automatically pulls a Java (Spring Boot) project from GitHub, compiles it using Maven, and triggers the build process automatically on every code commit.

Objectives:

1. To integrate **Apache Maven** with Jenkins for automated project building.
2. To configure **GitHub Webhooks** for automated build triggers (Continuous Integration).
3. To understand the **Build Life Cycle** of a Java application in a Master-Worker environment.

Theory:

1. Pipeline as Code (The Jenkinsfile) A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control. This practice is known as "Pipeline as Code". It allows the build process to be versioned and reviewed just like the application code. By placing this file in the **root directory** of the repository, Jenkins can automatically discover the build stages, making the pipeline portable and easy to replicate across different environments.

2. Apache Maven Build Lifecycle Maven is a powerful build automation tool primarily used for Java projects, operating on the concept of a Project Object Model (POM). It simplifies the build process by providing a uniform build system and managing complex project dependencies through its central repository. The standard build lifecycle consists of several phases, including compile, test, package, and install. When we execute the command mvn clean package, Maven first removes any previous build artifacts to ensure a fresh environment, then compiles the source code, runs unit tests (unless skipped), and finally packages the compiled code into a distributable format like a JAR or WAR file. This automation is vital in CI/CD pipelines because it ensures that every build is consistent, repeatable, and follows a standardized structure regardless of the environment it is running in.

3. Continuous Integration (CI) and Triggers Continuous Integration is a DevOps practice where developers frequently merge their code changes into a central repository, after which automated builds and tests are run. A "Trigger" is the specific mechanism that tells Jenkins to start this

process. While manual triggers are useful for testing, automated triggers are the heart of a true CI pipeline. By automating the build trigger, teams can identify bugs and integration issues much earlier in the development cycle, leading to higher software quality and faster release cadences.

4. **GitHub Webhooks and Public Accessibility** A Webhook is a lightweight HTTP callback that allows one application (GitHub) to send real-time data to another application (Jenkins) whenever a specific event occurs. For this communication to be successful, the Jenkins Master must have a **Publicly Accessible URL**, meaning it can be reached over the internet by GitHub's servers. When a developer commits code, GitHub sends a POST request to this public URL, which Jenkins interprets as a signal to pull the latest code and start the execution of the defined pipeline stages.

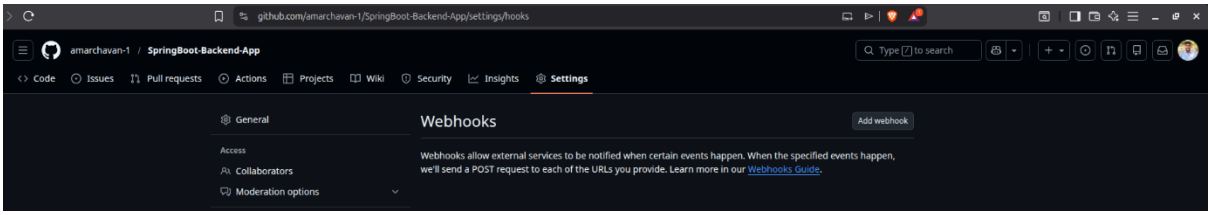
5. **Master-Worker Execution Logic** In a professional Jenkins setup, the Master (Controller) node handles the management tasks while the Worker node performs the heavy lifting, such as compilation and testing. This distributed architecture is crucial for Java projects because the compilation process can be resource-intensive, requiring significant CPU and RAM. By offloading the Maven build to a Worker node, the Master node remains responsive for other developers and administrative tasks, ensuring the overall CI/CD infrastructure is scalable and efficient.

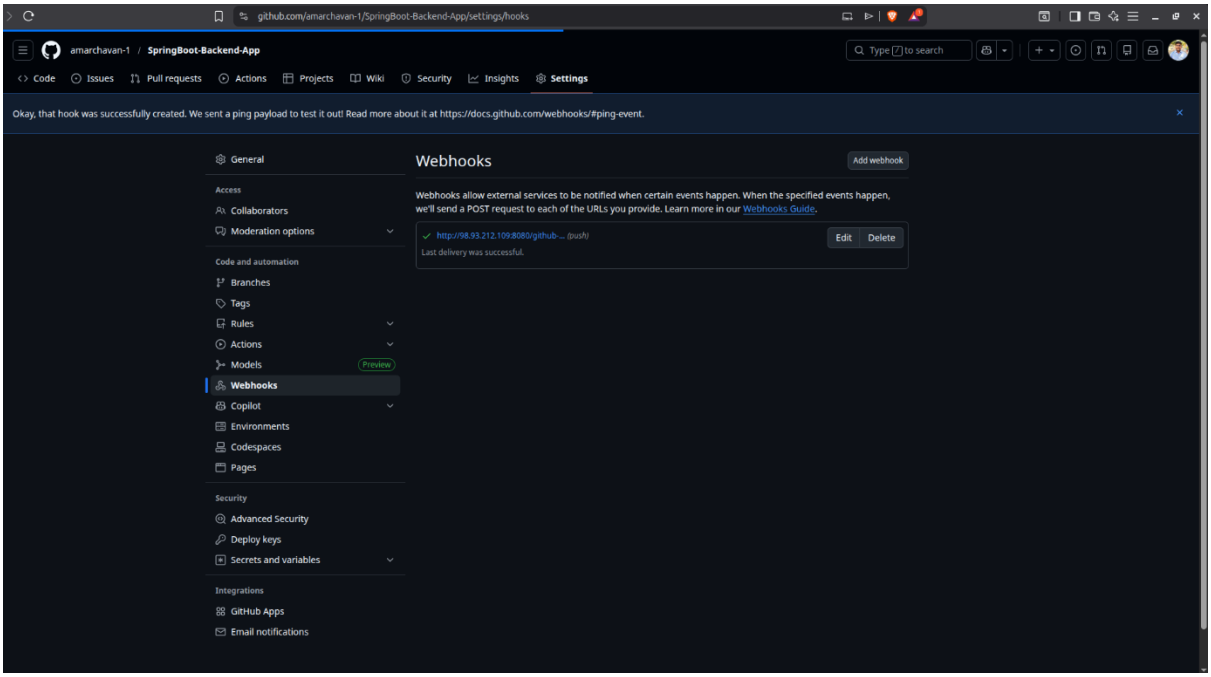
6. **Artifact Management and Workspace** During the build process, Jenkins creates a dedicated **Workspace** on the Worker node where the source code is cloned and the build is executed. Once the Maven build successfully completes, the resulting JAR file is stored in the target/ directory within this workspace. These artifacts represent the final, "ready-to-deploy" version of the software. Proper management of these artifacts is essential for tracking version history and ensuring that the exact code that was tested is the same code that eventually moves into the deployment phase.

Practical Procedure / Steps:

Step 1: Configure GitHub Webhook

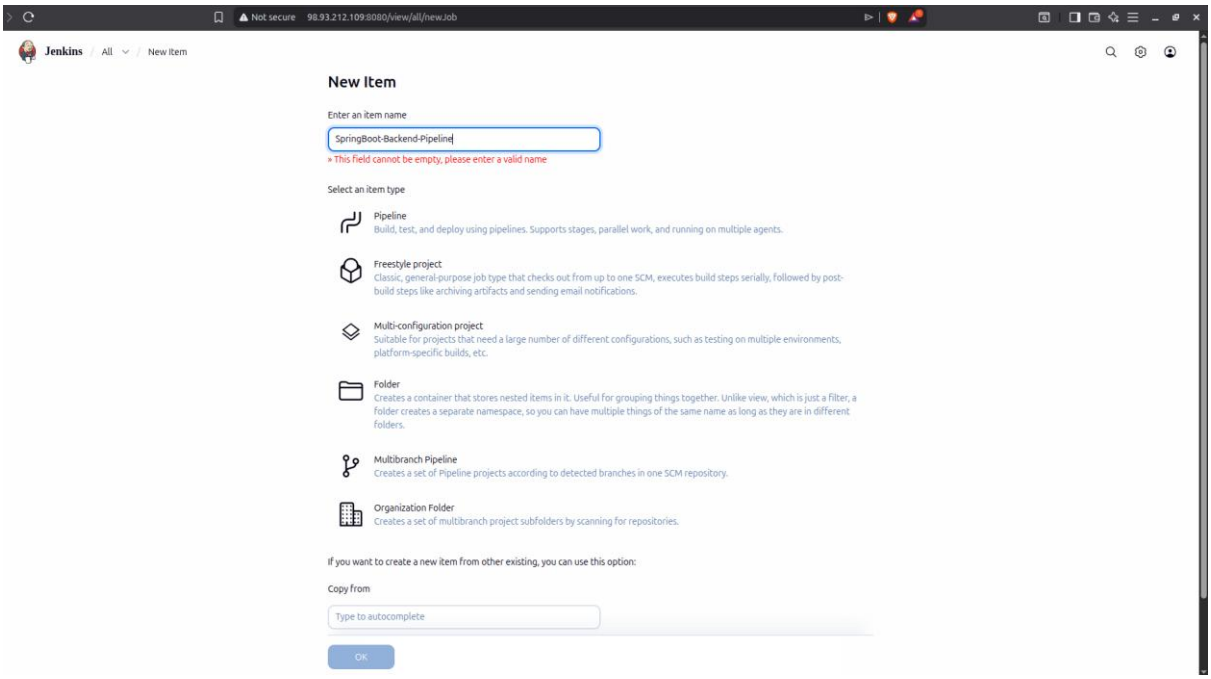
1. Navigate to your GitHub Repository **Settings > Webhooks**.
2. Click **Add webhook**.
3. **Payload URL:** http://<Jenkins-Master-Public-IP>:8080/github-webhook/.
4. **Content type:** application/json.
5. Select "Just the push event" and click **Add webhook**. Ensure a green checkmark appears.

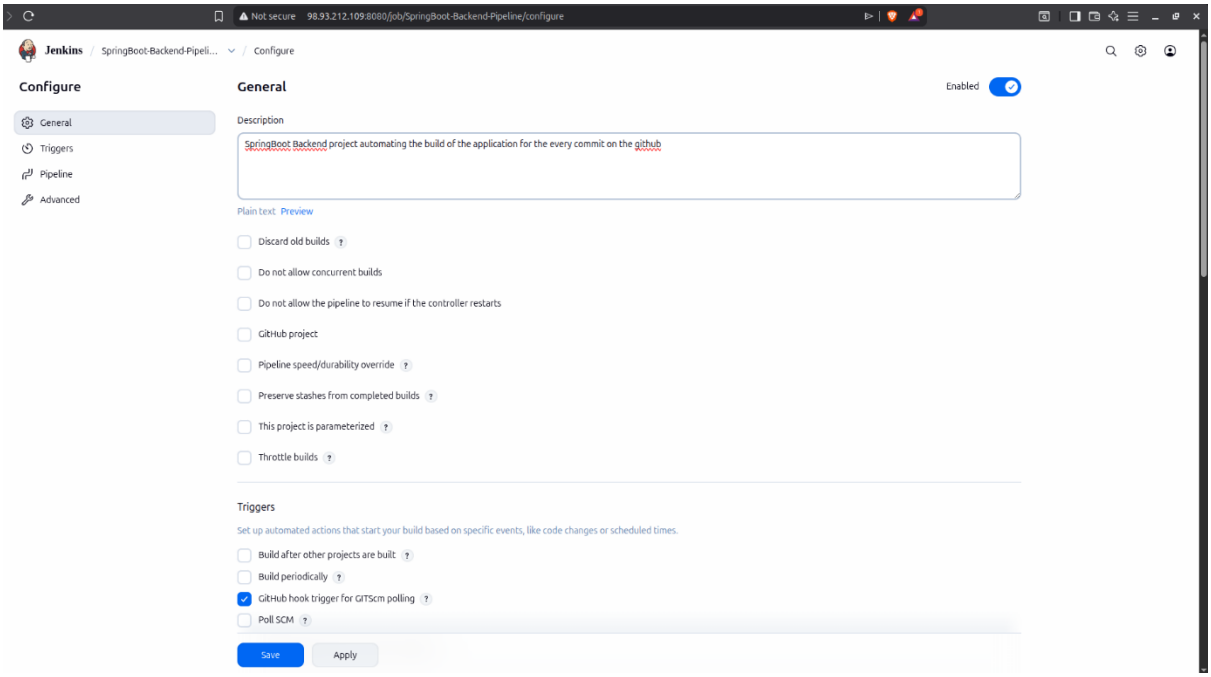


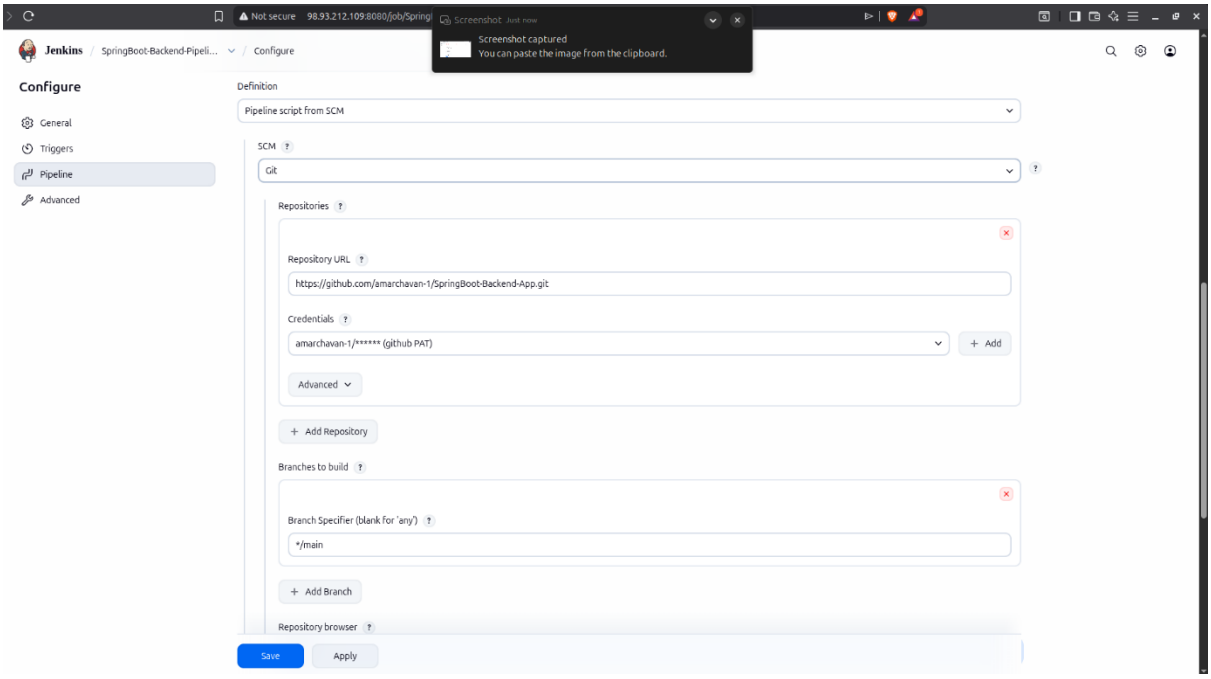


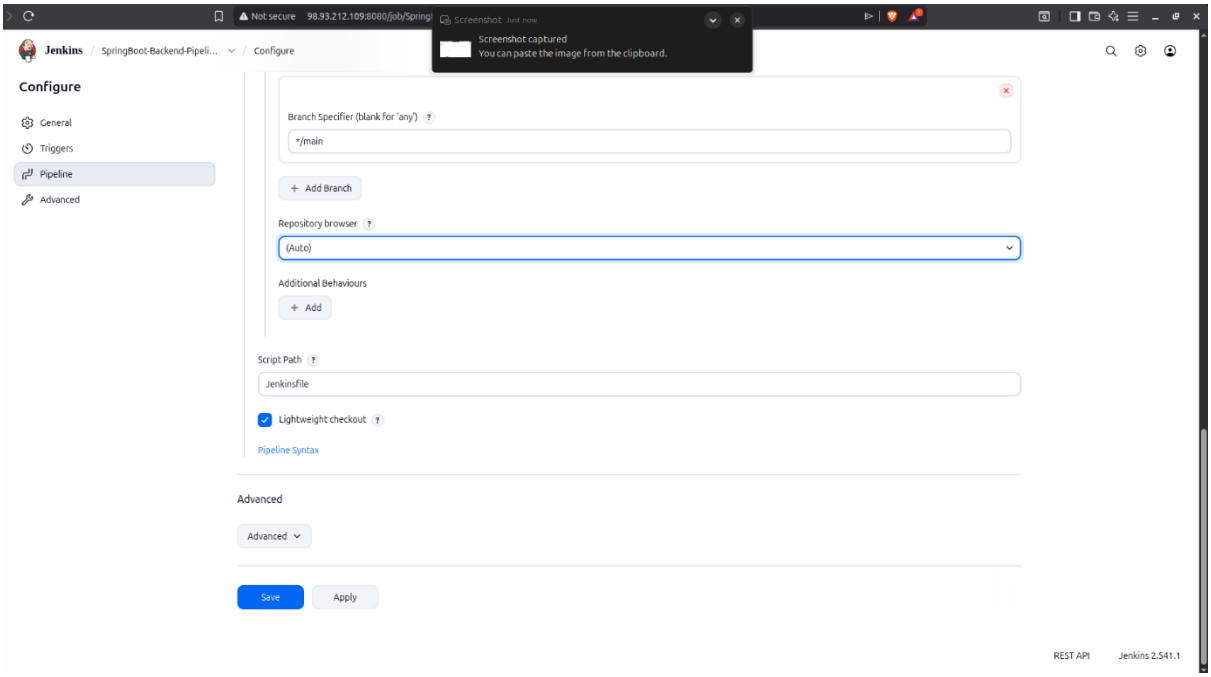
Step 2: Create a New Pipeline Job

1. Select **New Item** > **SpringBoot-Backend-Pipeline** > **Pipeline**.
2. Under **Build Triggers**, check the box: **"GitHub hook trigger for GITScm polling"**.



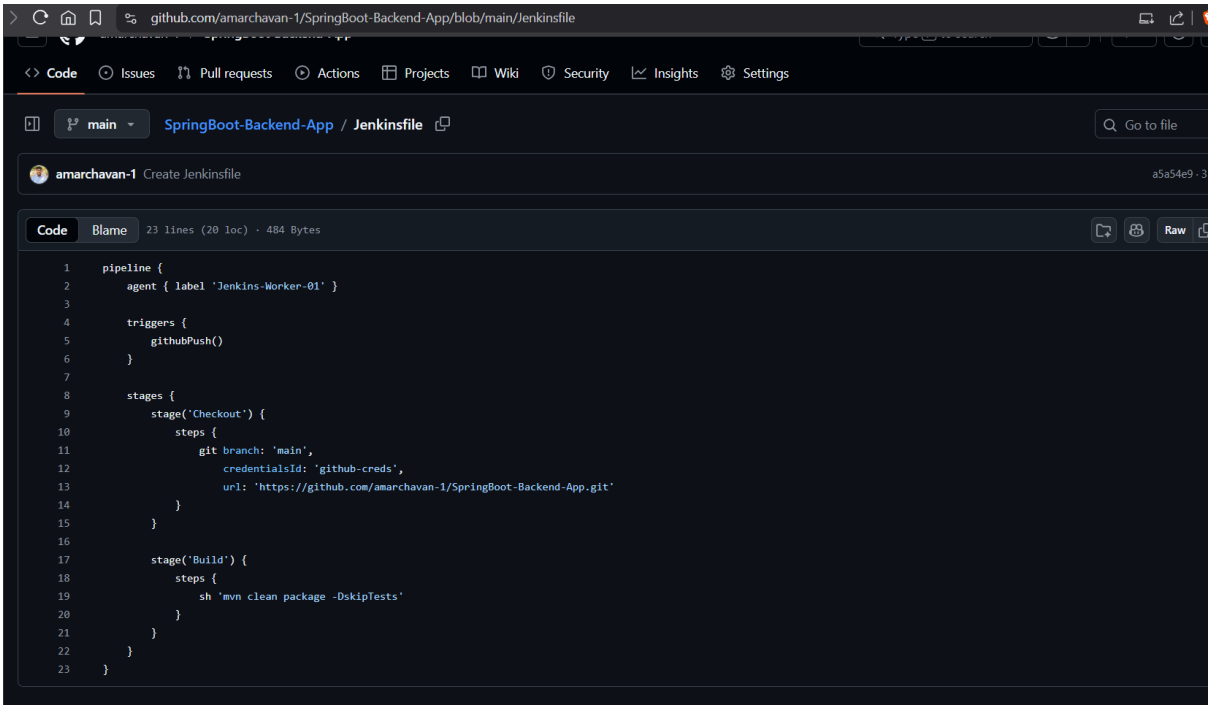

Not secure 98.93.212.109:8080/job/SpringBoot-Backend-Pipeline/configure
Jenkins / SpringBoot-Backend-Pipeli... / Configure
Configure
General
Triggers
Pipeline
Advanced
General
Enabled
Description
SpringBoot Backend project automating the build of the application for the every commit on the github
Plain text Preview
Discard old builds
Do not allow concurrent builds
Do not allow the pipeline to resume if the controller restarts
GitHub project
Pipeline speed/durability override
Preserve stashes from completed builds
This project is parameterized
Throttle builds
Triggers
Set up automated actions that start your build based on specific events, like code changes or scheduled times.
Build after other projects are built
Build periodically
GitHub hook trigger for GITScm polling
Poll SCM
Save
Apply


Not secure 98.93.212.109:8080/job/Spring
Screenshot Just now
Screenshot captured
You can paste the image from the clipboard.
Jenkins / SpringBoot-Backend-Pipeli... / Configure
Configure
General
Triggers
Pipeline
Advanced
Definition
Pipeline script from SCM
SCM
Git
Repositories
Repository URL
https://github.com/amarchavan-1/SpringBoot-Backend-App.git
Credentials
amarchavan-1/****** (github PAT)
Add
Advanced
Add Repository
Branches to build
Branch Specifier (blank for 'any')
*/main
Add Branch
Repository browser
Save
Apply



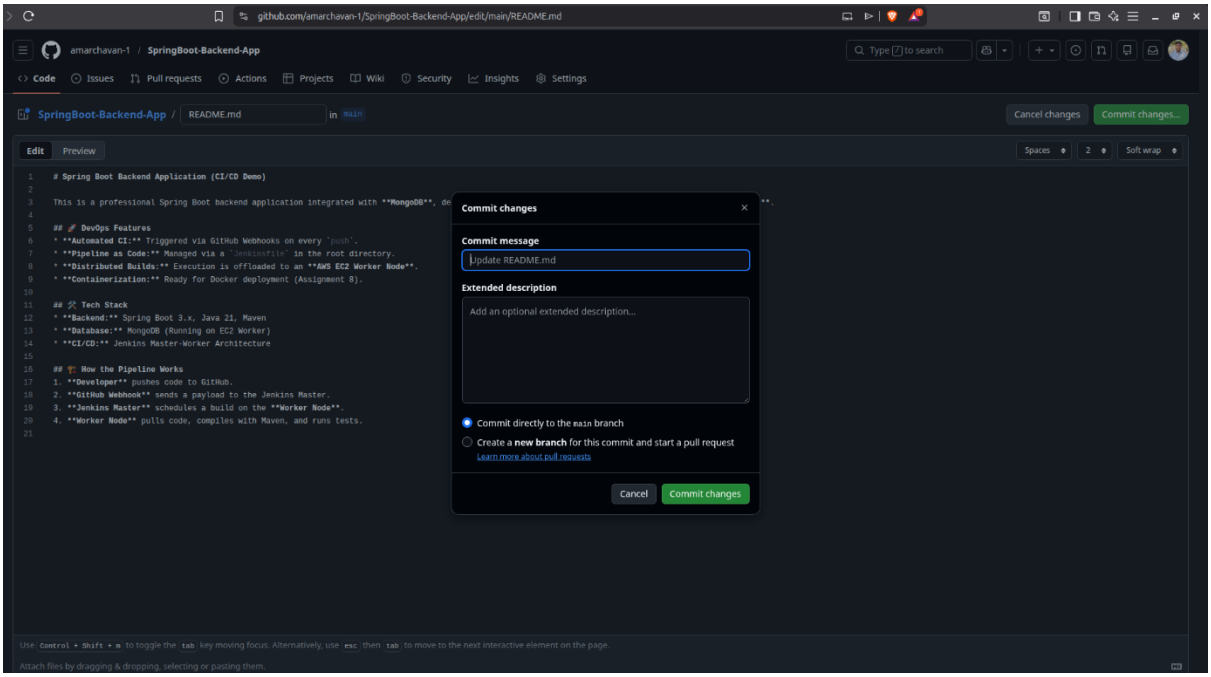
Step 3: Create the Jenkinsfile in GitHub

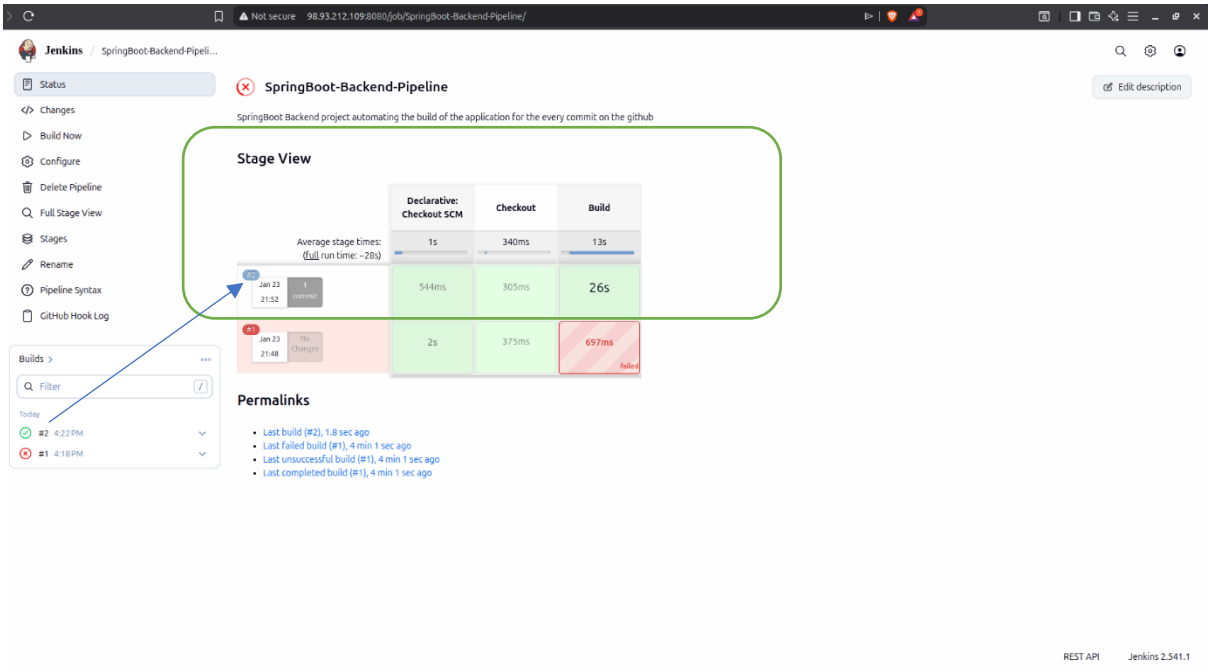
- Navigate to your GitHub repository root (SpringBoot-Backend-App).
- Create a new file named exactly Jenkinsfile (no extension).
- Paste the following declarative script:

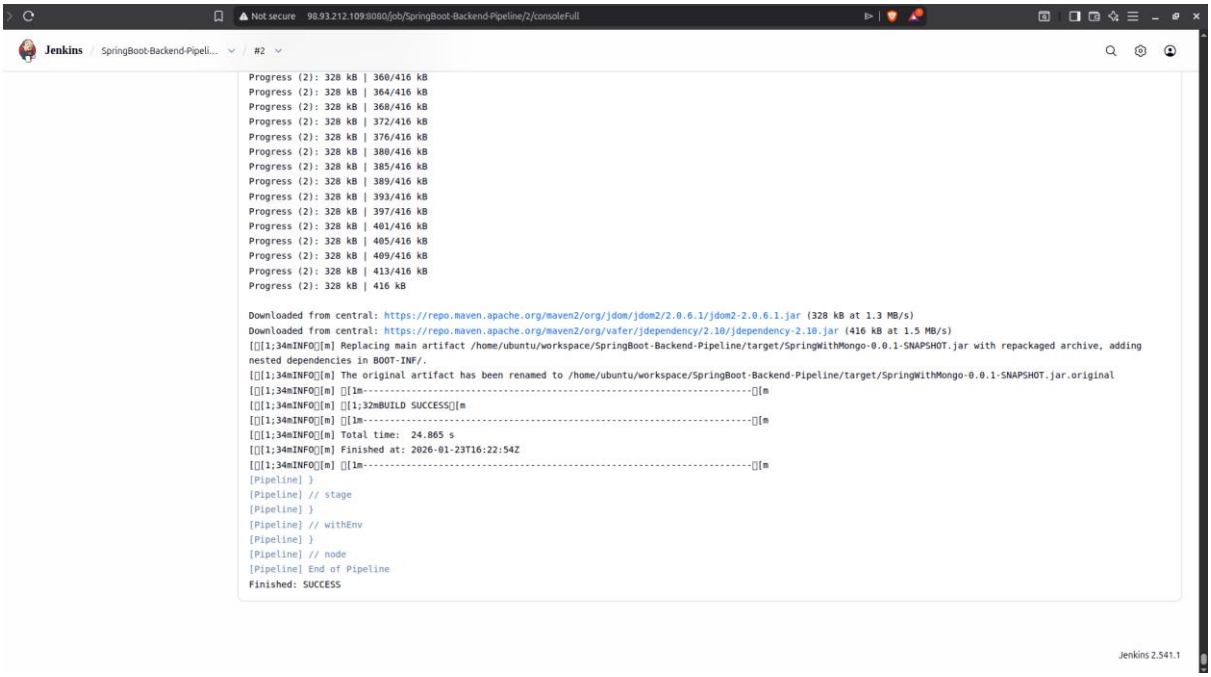


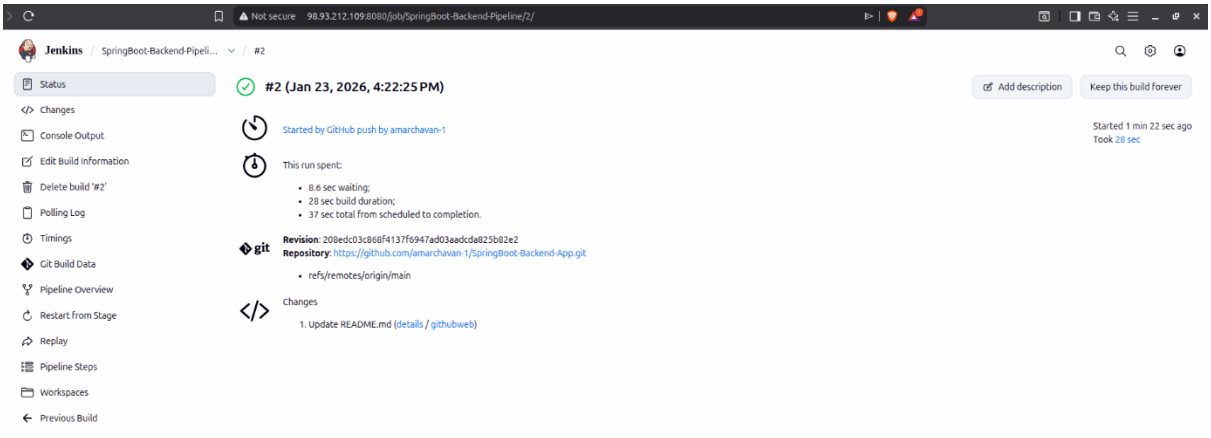
Step 4: Trigger the Automated Build

1. Go to GitHub and edit any file (e.g., change a line in README.md or a comment in a Java file).
2. **Commit** the changes.
3. Immediately switch to the **Jenkins UI**. You will observe that a new build (e.g., Build #2) has started **automatically** without any manual click.









Conclusion

The successful completion of this assignment demonstrates the powerful synergy between **Pipeline as Code** and **Automated Triggers**, which are the core pillars of modern DevOps practices. By implementing a Jenkinsfile directly within the GitHub repository root, we achieved a portable and version-controlled build process that remains synchronized with the application code. The integration of **GitHub Webhooks** proved to be a critical success, transforming a manual, error-prone task into a "zero-touch" **Continuous Integration (CI)** workflow. Every code commit now triggers an immediate, automated compilation on the **Worker Node**, providing developers with rapid feedback on build stability and code quality.

Furthermore, offloading the **Maven build lifecycle**—including compilation and JAR packaging—to a dedicated execution agent confirmed the efficiency of the **Master-Worker architecture**. This setup ensures the Master node remains responsive while the Worker handles the resource-intensive tasks. Ultimately, this assignment validates that the infrastructure is fully capable of supporting professional, enterprise-grade software development cycles, ensuring that the software is always in a buildable and verifiable state.