



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.

The top screenshot shows the 'Webhooks' section of the GitHub settings for the repository 'SpringBoot-Backend-App'. It includes a brief description of what webhooks do and a 'Add webhook' button.

The bottom screenshot shows the same 'Webhooks' section after a webhook has been successfully created. It lists the URL 'http://98.93.212.109:8080/github...' and indicates that the 'Last delivery was successful'.

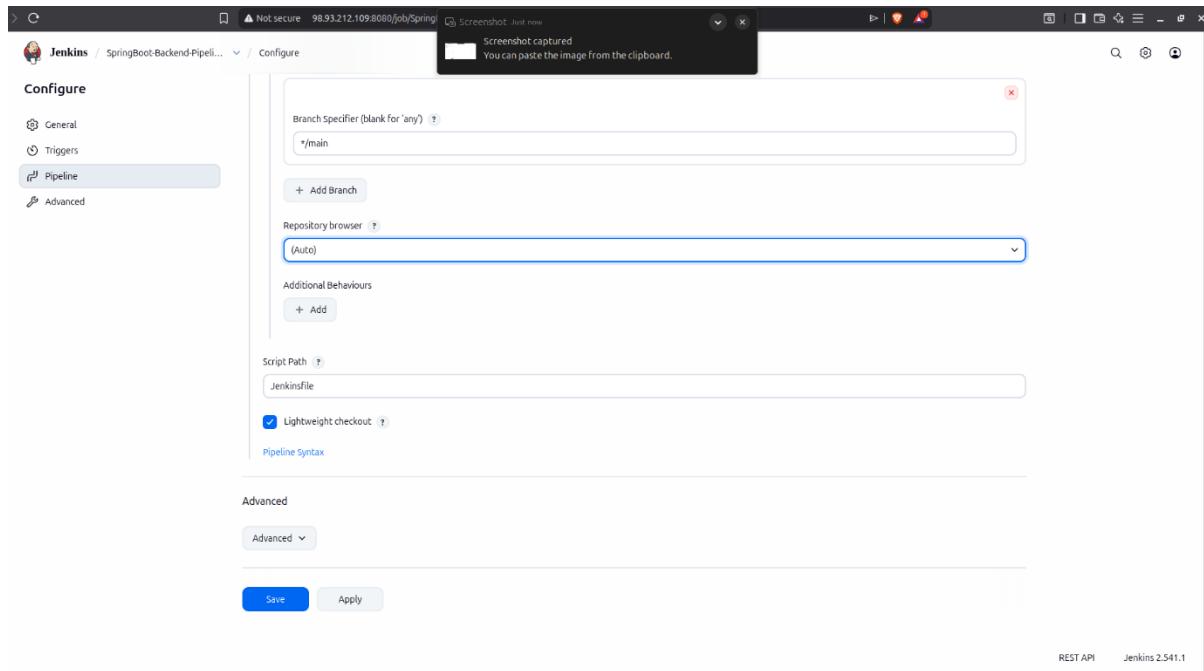
Step 2: Create a New Pipeline Job

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

The screenshot shows the Jenkins 'New Item' configuration page. The item name is set to 'SpringBoot-Backend-Pipeline'. The item type is selected as 'Pipeline'. The 'Build Triggers' section is expanded, showing the checkbox for 'GitHub hook trigger for GITScm polling' is checked. Other options like 'Poll SCM' and 'Build Periodically' are also visible.

The screenshot shows the Jenkins 'Configure' screen for a job named 'SpringBoot-Backend-Pipeline'. The 'General' tab is selected. In the 'Description' field, the text 'SpringBoot Backend project automating the build of the application for every commit on the github' is entered. Below the description, there are several configuration options: 'Discard old builds', 'Do not allow concurrent builds', 'Do not allow the pipeline to resume if the controller restarts', 'GitHub project' (which is checked), 'Pipeline speed/durability override', 'Preserve stashes from completed builds', 'This project is parameterized', and 'Throttle builds'. A note at the bottom of this section states: 'Set up automated actions that start your build based on specific events, like code changes or scheduled times.' Under the 'Triggers' section, 'GitHub hook trigger for GITScm polling' is checked. At the bottom of the page are 'Save' and 'Apply' buttons.

The screenshot shows the Jenkins 'Configure' screen for the same job, with the 'Pipeline' tab selected. In the 'Definition' section, 'Pipeline script from SCM' is chosen. Under the 'SCM' section, 'Git' is selected. The 'Repositories' section contains one repository with the URL 'https://github.com/amarchavan-1/SpringBoot-Backend-App.git' and a credential entry 'amarchavan-1/******** (github PAT)'. Below this, under 'Branches to build', the 'Branch Specifier (Blank for 'any')' field contains '/main'. At the bottom of the page are 'Save' and 'Apply' buttons.



Step 3: Create the Jenkinsfile in GitHub

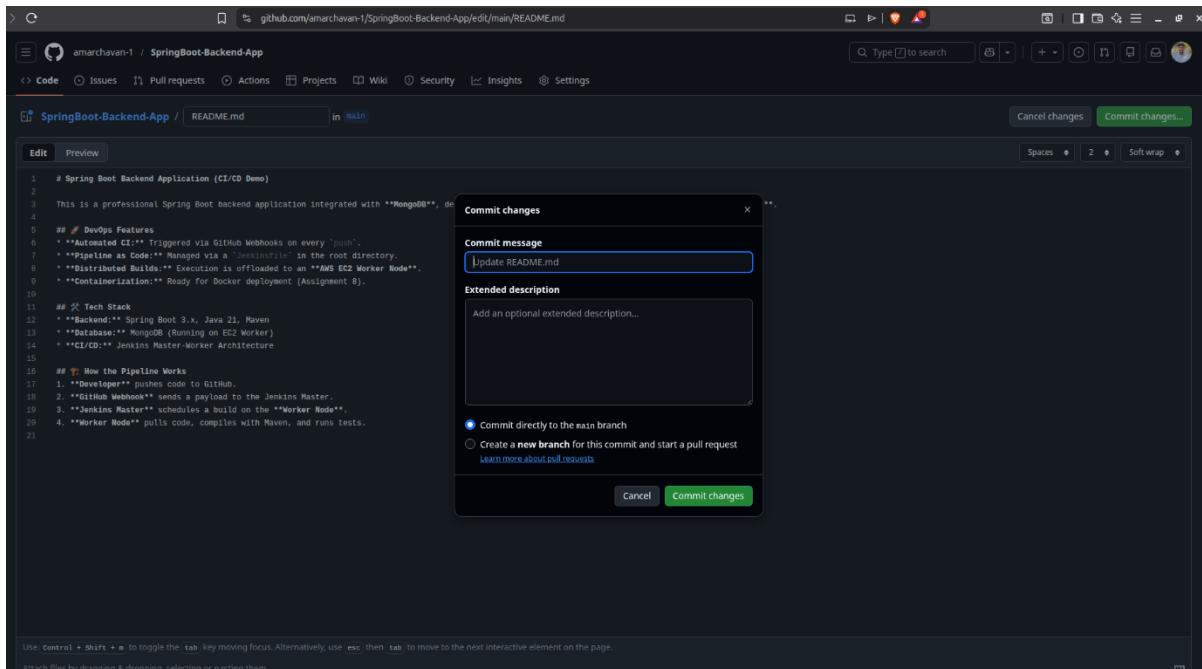
- Navigate to your GitHub repository root (SpringBoot-Backend-App).
- Create a new file named exactly Jenkinsfile (no extension).
- Paste the following declarative script:

A screenshot of a GitHub repository page for 'SpringBoot-Backend-App'. The 'Code' tab is selected, showing the Jenkinsfile. The code content is:

```
1 pipeline {
2     agent { label 'Jenkins-Worker-01' }
3 
4     triggers {
5         githubPush()
6     }
7 
8     stages {
9         stage('Checkout') {
10            steps {
11                git branch: 'main',
12                    credentialsId: 'github-creds',
13                    url: 'https://github.com/amarchavan-1/SpringBoot-Backend-App.git'
14            }
15        }
16 
17        stage('Build') {
18            steps {
19                sh 'mvn clean package -DskipTests'
20            }
21        }
22    }
23 }
```

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.



A screenshot of the Jenkins Pipeline interface for the 'SpringBoot-Backend-Pipeline'. The pipeline is described as 'SpringBoot Backend project automating the build of the application for every commit on the github'. The 'Stage View' section is highlighted with a green rounded rectangle and a blue arrow pointing from the GitHub commit message above. The stage view displays three stages: 'Declarative: Checkout SCM' (1s), 'Checkout' (340ms), and 'Build' (13s). Below the stages, a table shows execution times for specific commits: Jan 23 21:52 (1 commit, 544ms), Jan 23 21:48 (No Changes, 2s), and Jan 23 21:48 (#1 failed, 697ms). The Jenkins sidebar on the left includes options like Status, Changes, Build Now, Configure, Delete Pipeline, Full Stage View, Stages, Rename, Pipeline Syntax, and GitHub Hook Log. The bottom section shows a list of builds: #2 (4:22PM) and #1 (4:18PM). A 'Permalinks' section lists recent builds. The bottom right corner shows 'REST API' and 'Jenkins 2.541.1'.

```
Progress (2): 328 kB | 360/416 kB
Progress (2): 328 kB | 364/416 kB
Progress (2): 328 kB | 368/416 kB
Progress (2): 328 kB | 372/416 kB
Progress (2): 328 kB | 376/416 kB
Progress (2): 328 kB | 380/416 kB
Progress (2): 328 kB | 385/416 kB
Progress (2): 328 kB | 389/416 kB
Progress (2): 328 kB | 393/416 kB
Progress (2): 328 kB | 397/416 kB
Progress (2): 328 kB | 401/416 kB
Progress (2): 328 kB | 405/416 kB
Progress (2): 328 kB | 409/416 kB
Progress (2): 328 kB | 413/416 kB
Progress (2): 328 kB | 416 kB

Downloaded from central: https://repo.maven.apache.org/maven2/org/jdom/jdom2/2.0.6.1/jdom2-2.0.6.1.jar (328 kB at 1.3 MB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/vafer/dependency/2.10/dependency-2.10.jar (416 kB at 1.5 MB/s)
[::1:34]INFO[[m] Replacing main artifact /home/ubuntu/workspace/SpringBoot-Backend-Pipeline/target/SpringWithMongo-0.0.1-SNAPSHOT.jar with repackaged archive, adding nested dependencies in BOOT-INF/
[::1:34]INFO[[m] The original artifact has been renamed to /home/ubuntu/workspace/SpringBoot-Backend-Pipeline/target/SpringWithMongo-0.0.1-SNAPSHOT.jar.original
[::1:34]INFO[[m] -----[[m
[::1:34]INFO[[m] [::1:32]BUILD SUCCESS[[m
[::1:34]INFO[[m] [::1:-----[[m
[::1:34]INFO[[m] Total time: 24.865 s
[::1:34]INFO[[m] Finished at: 2026-01-23T16:22:54Z
[::1:34]INFO[[m] -----[[m
[Pipeline]
[Pipeline] // stage
[Pipeline]
[Pipeline] // withEnv
[Pipeline]
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```



The screenshot shows the Jenkins pipeline interface for build #2. The top navigation bar indicates the URL as 98.93.212.109:8080/job/SpringBoot-Backend-Pipeline/2/. The left sidebar contains links for Status, Changes, Console Output, Edit Build Information, Delete build '#2', Polling Log, Timings, Git Build Data, Pipeline Overview, Restart from Stage, Replay, Pipeline Steps, Workspaces, and Previous Build. The main content area displays the build status as 'Success' (#2 Jan 23, 2026, 4:22:25 PM), started by GitHub push by amarchavan-1. It shows a summary of run time spent (8.6 sec waiting, 28 sec build duration, 37 sec total from scheduled to completion). A git section details the revision (208edc02c06f413f6947ad03aadcd8a825b82e2) and repository (https://github.com/amarchavan-1/SpringBoot-Backend-App.git). The changes section lists one update to README.md. On the right, there are buttons for 'Add description' and 'Keep this build forever'. The bottom right corner shows the build took 1 min 22 sec ago and 28 sec.

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.