

Experiment 03

Aim:

To implement Continuous Integration (CI) using Jenkins by automating the build and test process for a Java project with Maven, triggered on every code change via GitHub.

Theory:

In modern software development, speed, stability, and collaboration are crucial. This is where **DevOps** practices like **Continuous Integration (CI)** come into play.

Continuous Integration (CI) is a software development practice where developers frequently merge their code changes into a central repository. Each integration is verified by an automated build and test process, allowing teams to detect problems early and improve the software's quality and delivery speed.

Jenkins is an open-source CI tool written in Java. It acts as an automation server that supports building, testing, and deploying software projects. Jenkins supports plugins that integrate with virtually every tool in the DevOps toolchain, from version control systems like Git to build tools like Maven and Gradle.

Maven is a build automation and dependency management tool for Java projects. It uses a file named pom.xml (Project Object Model) to manage project dependencies, build configuration, testing, and packaging.

GitHub is a popular web-based hosting service for version control using Git. By integrating GitHub with Jenkins, every code push or pull request can automatically trigger Jenkins to:

- Pull the updated source code
- Build the application using Maven
- Run test cases
- Generate reports
- Deploy artifacts

This experiment explores how Jenkins connects with GitHub and Maven to automate the software development lifecycle. It eliminates manual processes, reduces integration issues, and improves code stability.

Advantages of Jenkins CI Integration:

- Early detection of bugs and integration conflicts
- Reduced manual intervention and errors

- Real-time feedback on code quality
- Faster development and deployment cycles
- Supports scalable and repeatable processes across teams

REQUIREMENTS

- Java JDK (version 11 or above)
- Apache Maven
- Jenkins WAR package
- Git and GitHub repository
- GitHub Personal Access Token (for private repositories)
- Web browser
- Optional: ngrok (to expose localhost for webhook)

Procedure:

Part A: Environment and Tool Setup

1. Install Java JDK

- Verify Java: `java -version`

```
D:\Sem VII\DEVOPS\Experiment 03>java -version
java version "23.0.2" 2025-01-21
Java(TM) SE Runtime Environment (build 23.0.2+7-58)
Java HotSpot(TM) 64-Bit Server VM (build 23.0.2+7-58, mixed mode, sharing)
```

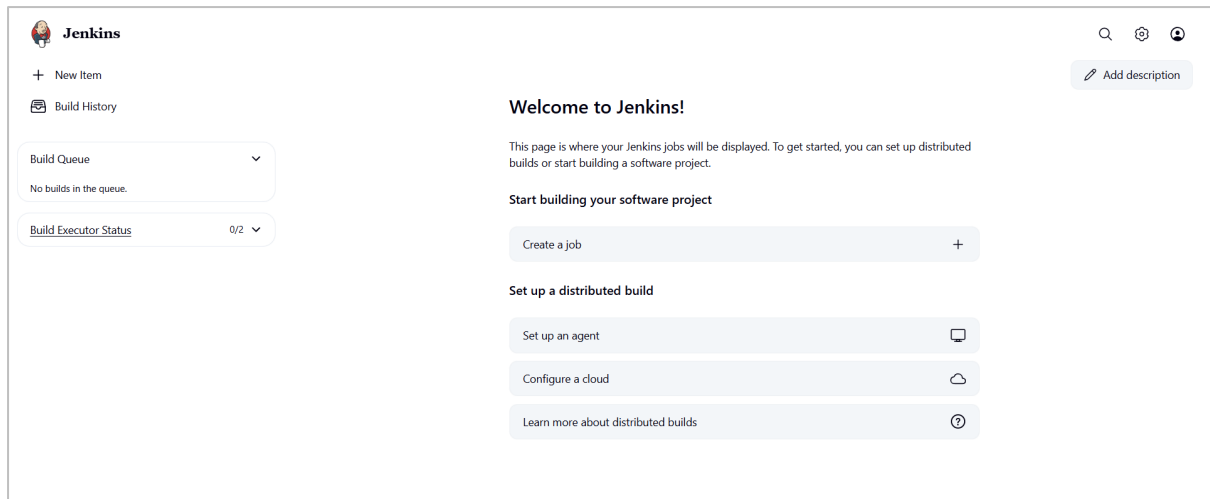
2. Install Jenkins (WAR method)

- Download Jenkins WAR:

```
>> wget http://mirrors.jenkins.io/war-stable/latest/jenkins.war
```

```
>> java -jar jenkins.war
```
- Open in browser: `http://localhost:8080`
- Unlock using initial password:

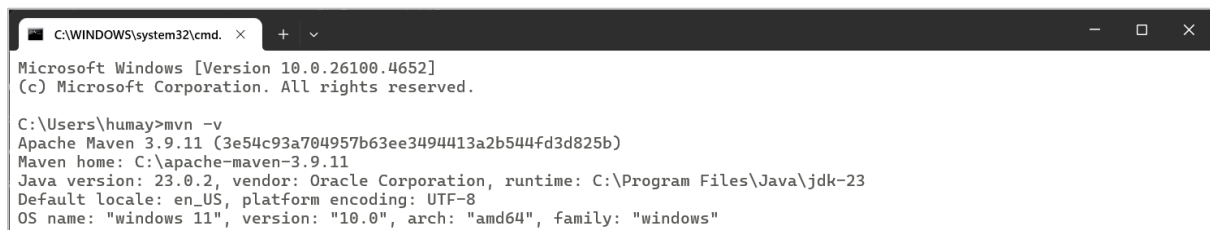
```
>> cat ~/.jenkins/secrets/initialAdminPassword
```
- Install Suggested Plugins
- Create an admin user



3. Install Apache Maven

- Download and unzip from: <https://maven.apache.org/download.cgi>
- Set MAVEN_HOME and update system PATH

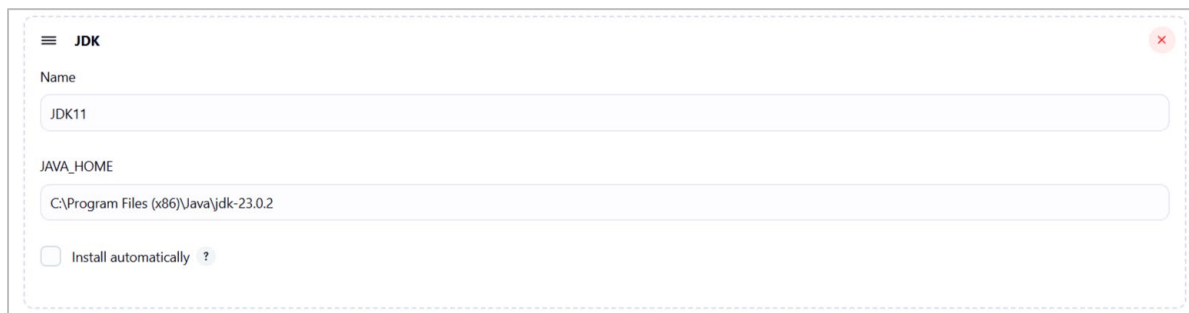
>> mvn -v



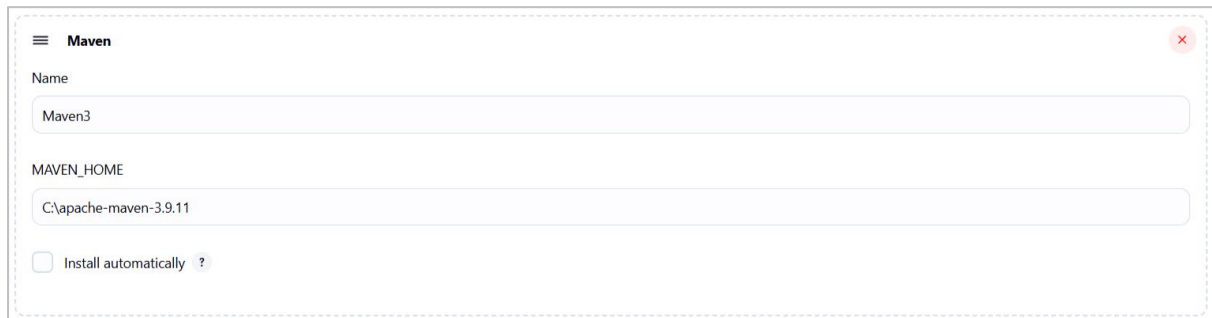
Part B: Configure Jenkins Tools

1. Global Tool Configuration

- Jenkins Dashboard → Manage Jenkins → Global Tool Configuration
- Add Java:
 - Name: JDK11
 - Uncheck "Install automatically"
 - Provide local JDK path



- Add Maven:
 - Name: Maven3
 - Uncheck “Install automatically”
 - Provide path from which mvn or Maven folder



The screenshot shows the 'Maven' configuration dialog in Jenkins. It has a title bar with a hamburger menu icon, the text 'Maven', and a close button (red X). The dialog contains two text input fields: the first is labeled 'Name' and contains the text 'Maven3'; the second is labeled 'MAVEN_HOME' and contains the text 'C:\apache-maven-3.9.11'. Below these fields is a checkbox labeled 'Install automatically' which is currently unchecked, followed by a small question mark icon.

2. Install Required Plugins

- Go to: Manage Jenkins → Plugin Manager → Available
- Install the following plugins:
 - Git Plugin
 - Maven Integration Plugin
 - GitHub Integration Plugin
 - JUnit Plugin
 - Pipeline Plugin (optional)
- Restart Jenkins if prompted

Part C: Connect Jenkins to GitHub

1. Fork or Create Maven Project

- Use a sample project like Spring PetClinic or create your own Java project with:
 - pom.xml
 - Source and test files

2. Set Up GitHub Webhook

- GitHub → Repository Settings → Webhooks → Add Webhook
 - Payload URL: `http://<your-ip>:8080/github-webhook/`
 - Content type: application/json
 - Event: Push events
- If Jenkins is hosted locally, use ngrok to expose:

```
>> ngrok http 8080
```

Webhooks / Manage webhook

[Settings](#)[Recent Deliveries](#)

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL ***Content type *****Secret****SSL verification**

 By default, we verify SSL certificates when delivering payloads.

☒ **Enable SSL verification** ☐ **Disable (not recommended)**

Which events would you like to trigger this webhook?

- ☒ Just the push event.
- ☐ Send me **everything**.
- ☐ Let me select individual events.

☒ **Active**

We will deliver event details when this hook is triggered.

[Update webhook](#)[Delete webhook](#)

3. Add GitHub Credentials in Jenkins



- Jenkins Dashboard → Manage Jenkins → Credentials → (Global) → Add Credentials
- Type: Username with Password
 - Username: GitHub username
 - Password: GitHub Personal Access Token
- Generate a token from:
 - GitHub → Settings → Developer Settings → Personal Access Tokens

- Scope: repo, workflow

Fine-grained personal access tokens

Generate new token

These are fine-grained, repository-scoped tokens suitable for personal [API](#) use and for using Git over HTTPS.

**Jenkins**
Never used •  This token has no expiration date



Delete

Jenkins / Manage Jenkins / Credentials / System / Global credentials (unrestricted)

Global credentials (unrestricted)

+ Add Credentials

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind	Description
 73857833-8f4b-4dbc-84c7-176ad33136b2	HumayunK01/*****	Username with password	

Icon: S M L

Part D: Create Jenkins Build Job

1. Create Freestyle Project

- Jenkins Dashboard → New Item → Name: MyJavaCIJob
- Select: Freestyle Project → OK


Jenkins / All / New Item


New Item


Enter an item name


MyJavaCIJob


Select an item type

 **Freestyle project**
Classic, general-purpose job type that checks out from up to one SCM, executes build steps serially, followed by post-build steps like archiving artifacts and sending email notifications.

 **Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

 **Pipeline**
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

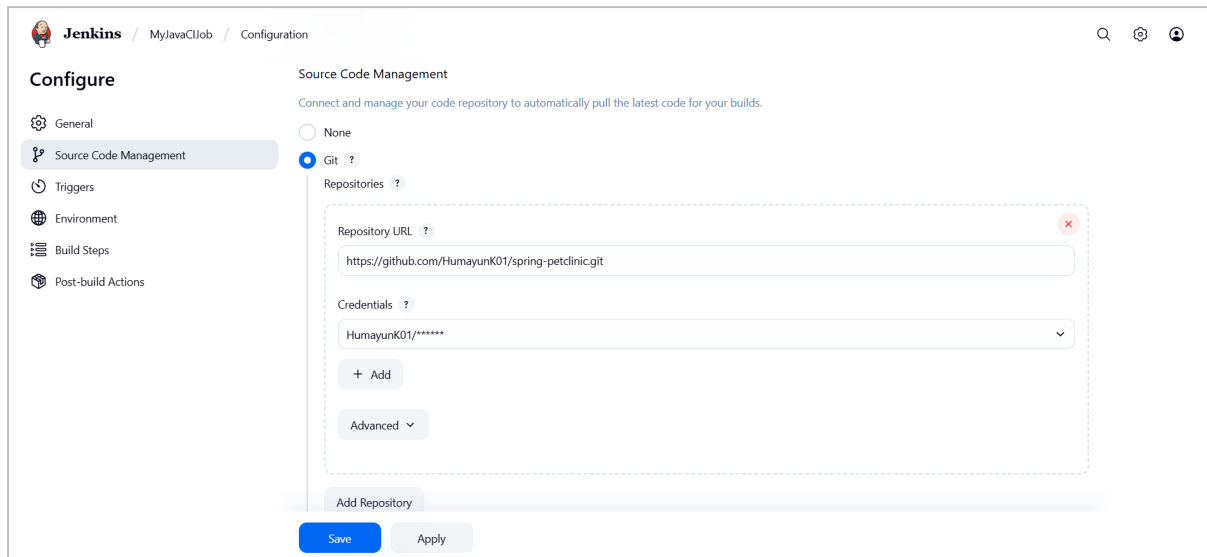
 **Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

 **Folder**

OK

2. Configure Source Code Management

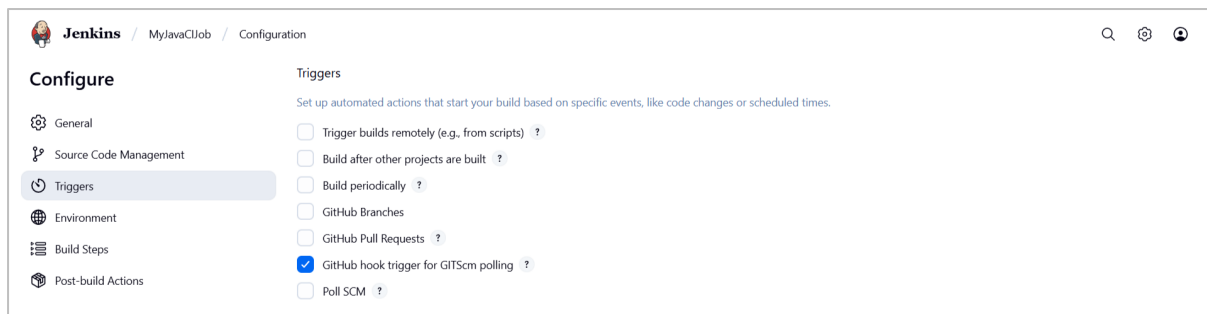
- Select Git
- Provide GitHub repository URL
- <https://github.com/HumayunK01/spring-petclinic.git>
- Add credentials (for private repos)



The screenshot shows the Jenkins configuration page for 'MyJavaCUIJob' under the 'Configuration' tab. The 'Source Code Management' section is active. It shows the 'Git' option selected under 'Source Code Management'. A repository is configured with the URL 'https://github.com/HumayunK01/spring-petclinic.git' and credentials 'HumayunK01/*****'. The 'Add Repository' button is visible at the bottom of the configuration area.

3. Set Build Trigger

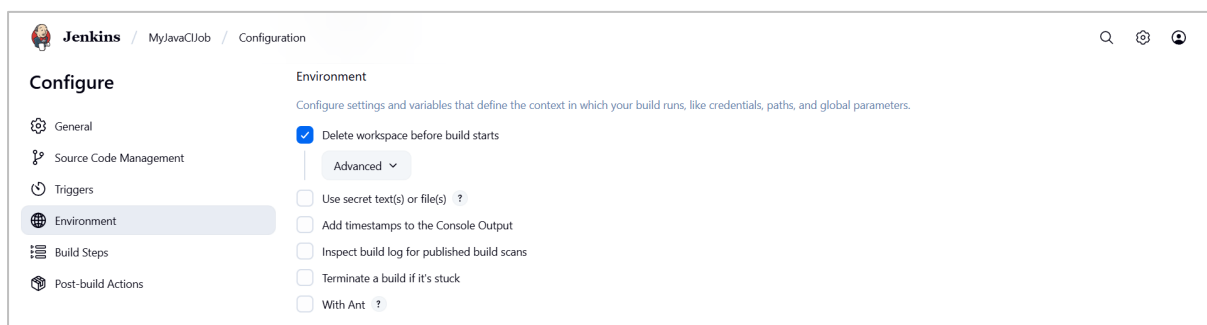
- Enable: GitHub hook trigger for GITScm polling



The screenshot shows the Jenkins configuration page for 'MyJavaCUIJob' under the 'Configuration' tab. The 'Triggers' section is active. It shows the 'GitHub hook trigger for GITScm polling' option checked, which enables the build trigger.

4. Configure Build Environment

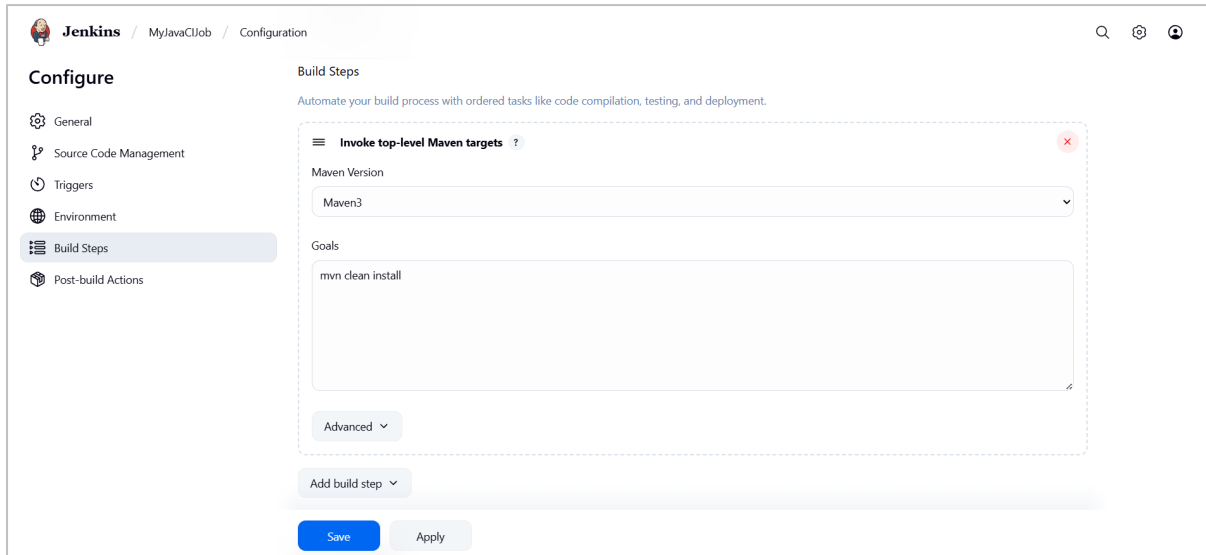
- Optionally enable: Delete workspace before build starts



The screenshot shows the Jenkins configuration page for 'MyJavaCUIJob' under the 'Configuration' tab. The 'Environment' section is active. It shows the 'Delete workspace before build starts' option checked, which enables the build environment configuration.

5. Add Build Steps

- Build Step: Invoke Maven Targets
 - Command:
>> mvn clean install



Jenkins / MyJavaCIJob / Configuration

Configure

- General
- Source Code Management
- Triggers
- Environment
- Build Steps**
- Post-build Actions

Build Steps

Automate your build process with ordered tasks like code compilation, testing, and deployment.

Invoke top-level Maven targets ?

Maven Version

Maven3

Goals

mvn clean install

Advanced

Add build step

Save Apply

Output:

- Java and Maven installation

```
C:\WINDOWS\system32\cmd. x + v
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\humay>java -version
java version "23.0.2" 2025-01-21
Java(TM) SE Runtime Environment (build 23.0.2+7-58)
Java HotSpot(TM) 64-Bit Server VM (build 23.0.2+7-58, mixed mode, sharing)

C:\Users\humay>mvn -v
Apache Maven 3.9.11 (3e54c93a704957b63ee3494413a2b544fd3d825b)
Maven home: C:\apache-maven-3.9.11
Java version: 23.0.2, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-23
Default locale: en_US, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"

C:\Users\humay>|
```

- Jenkins installation and startup

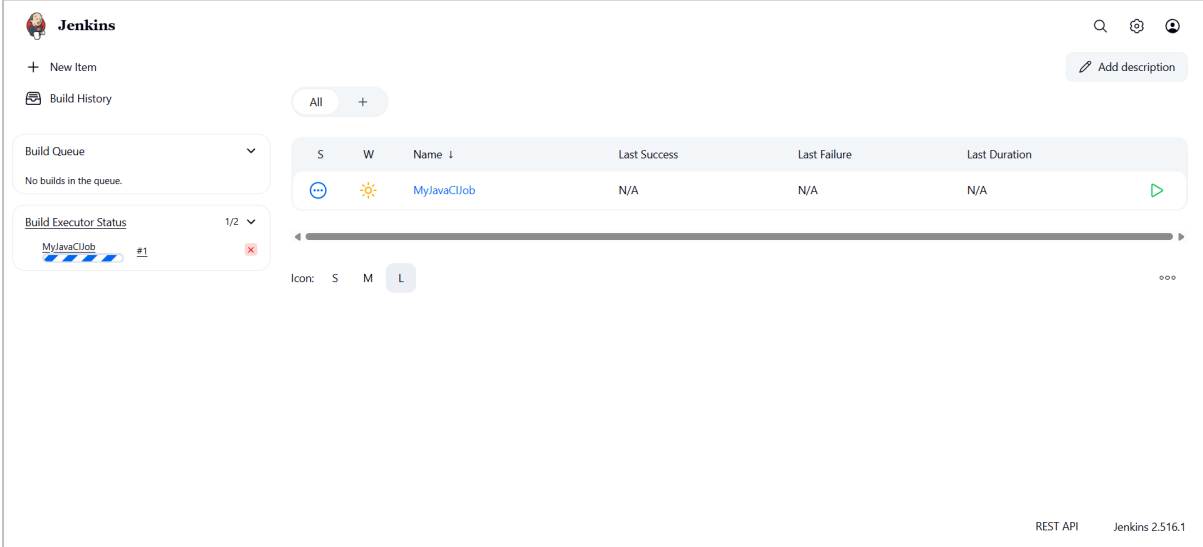
Getting Started

Jenkins is ready!

Your Jenkins setup is complete.

Start using Jenkins

- Jenkins dashboard with configured job



The screenshot shows the Jenkins dashboard. On the left, there's a sidebar with 'New Item' and 'Build History' links. Below them are sections for 'Build Queue' (empty) and 'Build Executor Status' (showing 1/2 executors). The main area features a table of jobs. The 'MyJavaCJob' job is highlighted with a sun icon. Below the table is a timeline for the selected job, showing it is currently running (indicated by a blue bar). The bottom right corner shows 'REST API' and 'Jenkins 2.516.1'.

S	W	Name ↓	Last Success	Last Failure	Last Duration
...	☀	MyJavaCJob	N/A	N/A	N/A

Icon: S M L

- GitHub webhook settings

Webhooks / Manage webhook

Settings

Recent Deliveries

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, `x-www-form-urlencoded`, etc). More information can be found in [our developer documentation](#).

Payload URL *

`https://1d1ab2fb7919.ngrok-free.app/github-webhook/`

Content type *

`application/json`

Secret

SSL verification



By default, we verify SSL certificates when delivering payloads.

☒ **Enable SSL verification** ☐ **Disable (not recommended)**

Which events would you like to trigger this webhook?

☒ Just the push event.

☐ Send me **everything**.

☐ Let me select individual events.

☒ **Active**

We will deliver event details when this hook is triggered.

Update webhook **Delete webhook**

- Build trigger and output logs

Jenkins

MyJavaCIJob

Configuration

Configure

General

Source Code Management

Triggers

Environment

Build Steps

Post-build Actions

Triggers

Set up automated actions that start your build based on specific events, like code changes or scheduled times.

☐ Trigger builds remotely (e.g., from scripts) ?

☐ Build after other projects are built ?

☐ Build periodically ?

☐ GitHub Branches

☐ GitHub Pull Requests ?

☒ GitHub hook trigger for GITScm polling ?

☐ Poll SCM ?

Conclusion:

This experiment successfully demonstrates the implementation of a continuous integration pipeline using Jenkins, Maven, and GitHub. By configuring Jenkins to automatically respond to code changes in a GitHub repository, developers can automate the build and testing process. This results in faster feedback, early bug detection, reduced integration issues, and improved development productivity. The integration of GitHub, Maven, and Jenkins embodies the core principles of DevOps in practice.