

What is Continuous Integration?

Continuous Integration is a development practice that requires developers to integrate code into a shared repository at regular intervals. The common practice is that whenever a code commit occurs, a build should be triggered. Small changes are committed more often rather than committing large changes once to trigger nightly builds.

Why Continuous Integration

While working in large teams, developers need to commit their code into the shared repository several times a day. Developers working on different files at the same time or on same files at different time zones need to pull changes made by others at regular intervals. However, these commits can occur at irregular intervals causing delays in integration leading to further delay in testing and deployment. Another issue with delayed integration is difficulty in isolation of defects. Since entire code is integrated at once developers have no idea what caused the build failure. This ultimately slows down the delivery process.

What is Jenkins?

Jenkins is a continuous integration tool that allows continuous development, test and deployment of newly created code, regardless of the platform you are working on. It is a free source that can handle any kind of build or continuous integration. You can integrate Jenkins with a number of testing and deployment technologies.

Jenkins can automate all sorts of tasks related to building, testing, and delivering or deploying software.

Benefits of Continuous Integration

Fast feedback loop: Lack of feedback on the quality and impact of the changes you made can be very expensive and slows down your delivery process. By not integrating and testing your code for small changes can make it difficult to test later if anything breaks. Continuous integration tools, when properly used, will remove much of this headache by providing you with quick answers to the question “did I break something?” for each commit.

Increase transparency and visibility: When your CI/CD pipeline is set up, your entire team will know what’s going on with the builds as well as get the latest results of tests, which means they can raise issues and plan their work in context.

Avoid Integration hell: Integrating the code developed by different team members is rarely a smooth process. If a specific piece is fine on its own in terms of implemented code, you will still need to make sure it plays nice with everything else. Continuous integration supports connecting the pieces of your software every day.

Improve quality and testability:

Sometimes the code works locally, however when the code is integrated things change and the tests that ran locally may fail. Continuous Integration tests your code against the current state of your code base and always in the same (production-like) environment. By running the tests more frequently and in controlled environment one can assure better quality.

Installation:

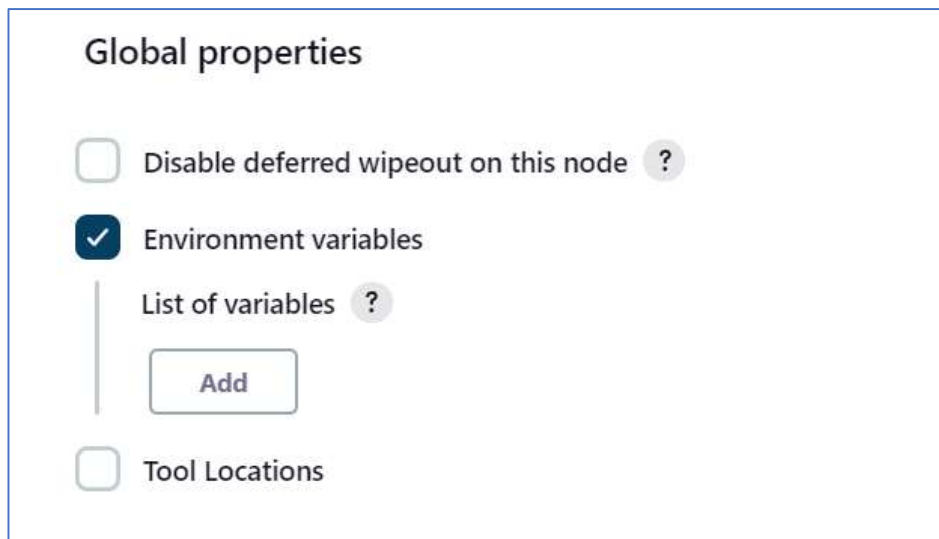
<https://www.jenkins.io/doc/book/installing/windows/>

Note down the admin key required to log in

Jenkins Global Properties

We can set global properties by navigating to “Manage Jenkins -> Configure System -> Global properties option

Check the “Environment variables” checkbox and then add the variables and their respective values inside the “List of Variables” section



The screenshot shows the 'Global properties' configuration page in Jenkins. It features three main sections, each with a checkbox and a help icon (a circle with a question mark):

- ☐ Disable deferred wipeout on this node ?
- ☒ Environment variables
 - Below this checkbox is a vertical line and the text 'List of variables ?'.
 - Below that is a button labeled 'Add'.
- ☐ Tool Locations

Global properties

☐ Disable deferred wipeout on this node ?

☒ Environment variables

List of variables ?

Name

JAVA_HOME

Value

C:\Program Files\Java\jdk-11.0.15.1

Save **Apply**

Name

MAVEN_HOME

Value

C:\Maven\apache-maven-3.8.6

Add

☐ Tool Locations

Setting Up Global Tool Configuration

Go to Manage Jenkins -> Global Tool Configuration

Make sure that configuration options are set for all plugins and tools. For instance if you have added Maven and Git plugins the your Global tool configuration should be similar as below:

JDK

JDK installations

List of JDK installations on this system

Add JDK

JDK

Name

jdk-11.0.15.1

JAVA_HOME

C:\Program Files\Java\jdk-11.0.15.1

☐ Install automatically ?

Add JDK

Git installations

Git

Name

Default

Path to Git executable ?

C:\Program Files\Git\bin\git.exe

☐ Install automatically ?

Add Git

Maven

Maven installations

List of Maven installations on this system

Add Maven

Maven

Name

MavenLocal

MAVEN_HOME

C:\Maven\apache-maven-3.8.6

☐ Install automatically ?

Add Maven

Creating Maven Project:

Save the settings by clicking on Apply button > Go back to Jenkins dashboard > Click on New Item > Enter the project name and select 'Maven Project' radio.

Go to Settings:

Source Code Management

☒ None

☐ Git ?

Build

Root POM ?

D:\UserA\DemoTesting\pom.xml

Goals and options ?

clean test

Advanced...

Save the project. It will be displayed on the Jenkins Dashboard. Click on Build now button to build the project.

Dashboard >

+ New Item

People

Build History

Project Relationship

Check File Fingerprint

Manage Jenkins

All +

Add description

S	W	Name ↓	Last Success	Last Failure	Last Duration
✓	☀	DemoProj2	5 min 14 sec #11	7 mo 16 days #8	6.5 sec
...	☀	DemoTest	N/A	N/A	N/A

Trigger a build by clicking arrow in front of the project. A green tick in front of the project will show the build is successful. [Refresh if required]

Click on the name of the project to see the build history:

Status

Changes

Workspace

Build Now

Configure

Delete Maven project

Modules

Rename

Maven project DemoTest

Latest Test Result (no failures)

Permalinks

- Last build (#1), 29 sec ago
- Last stable build (#1), 29 sec ago
- Last successful build (#1), 29 sec ago
- Last completed build (#1), 29 sec ago

Build History

trend

Filter builds...

#1

Aug 2, 2023, 7:27 PM

↑

Click on the build #1 to see build details.

Console output will show the results on console.

Continuous Integration

We are going to create a Jenkins Job which will deploy a build locally every time there is a push request.

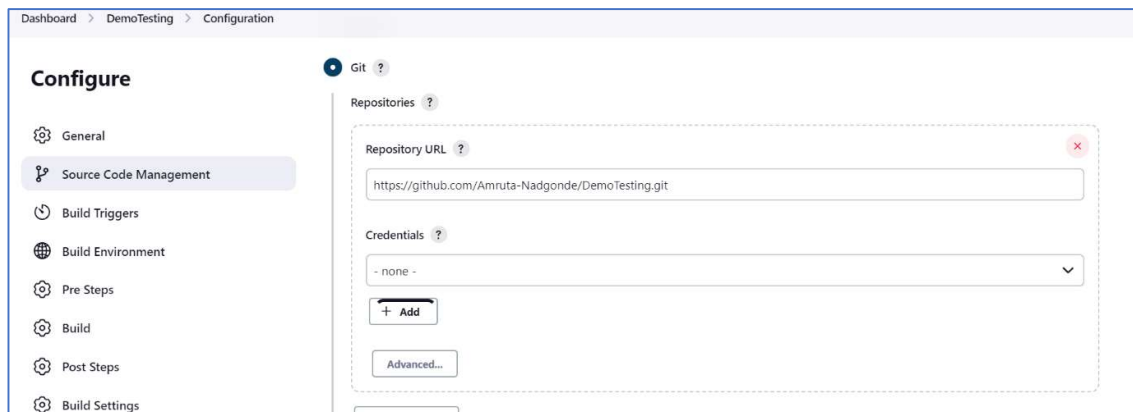
You can create similar jobs for Git pull requests or any other SCM events

Create a new Maven project in Jenkins

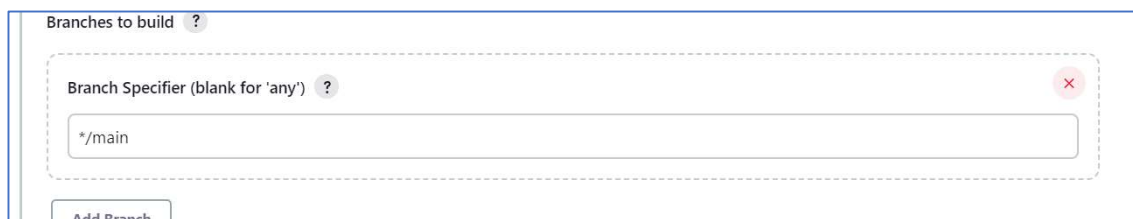
Jenkins -> New Item -> Enter Name of the Project -> Maven Project

In Source Code Management -> Select Git -> Enter Repo URL

In Branches to build -> select main

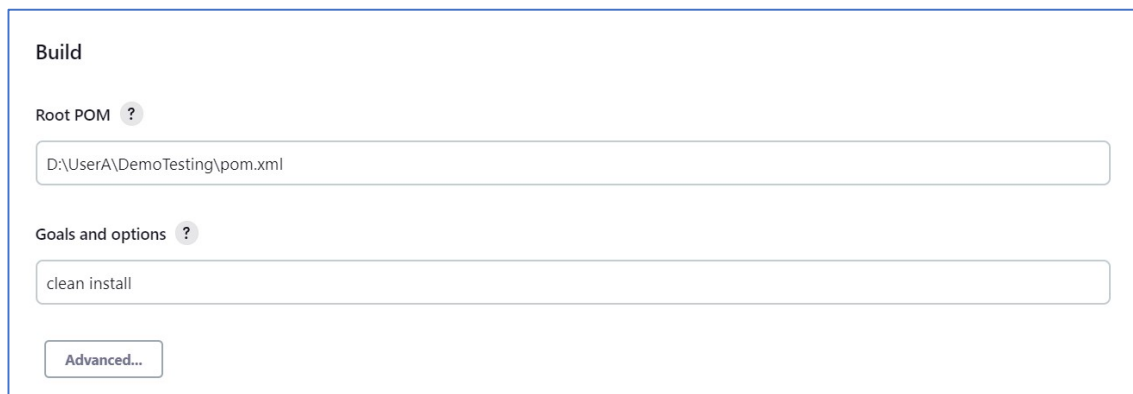


The screenshot shows the Jenkins 'Configure' page for a new item named 'DemoTesting'. The 'Source Code Management' section is selected in the left sidebar. Under 'Git', the 'Repository URL' is set to 'https://github.com/Amruta-Nadgonde/DemoTesting.git'. The 'Credentials' dropdown is set to '- none -'. There are '+ Add' and 'Advanced...' buttons below the credentials field.



The screenshot shows the 'Branches to build' section. The 'Branch Specifier (blank for 'any')' field is set to '*/main'. There is an 'Add Branch' button at the bottom left.

In build section -> Root POM -> POM -> Goals and Options -> Give maven command [e.g. clean install] -> Save



The screenshot shows the 'Build' section. The 'Root POM' field is set to 'D:\UserA\DemoTesting\pom.xml'. The 'Goals and options' field is set to 'clean install'. There is an 'Advanced...' button at the bottom left.

Now go to GitHub -> Select Repo -> Settings -> Webhooks

1. Click on AddWebhook [Webhooks allow external services to be notified when certain events happen]

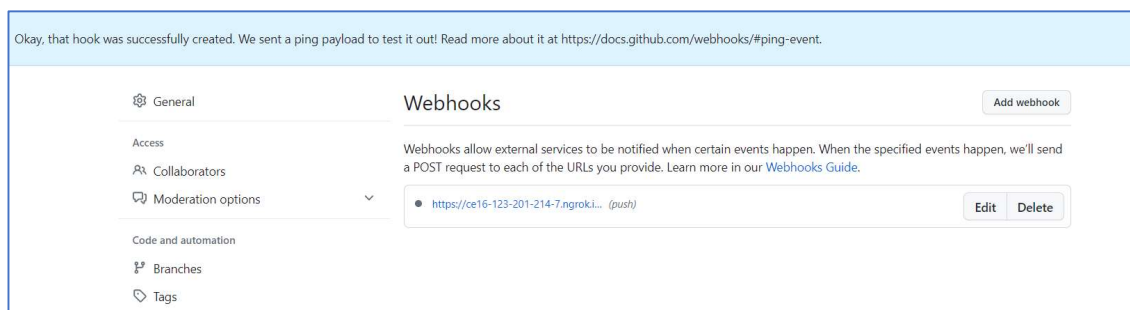
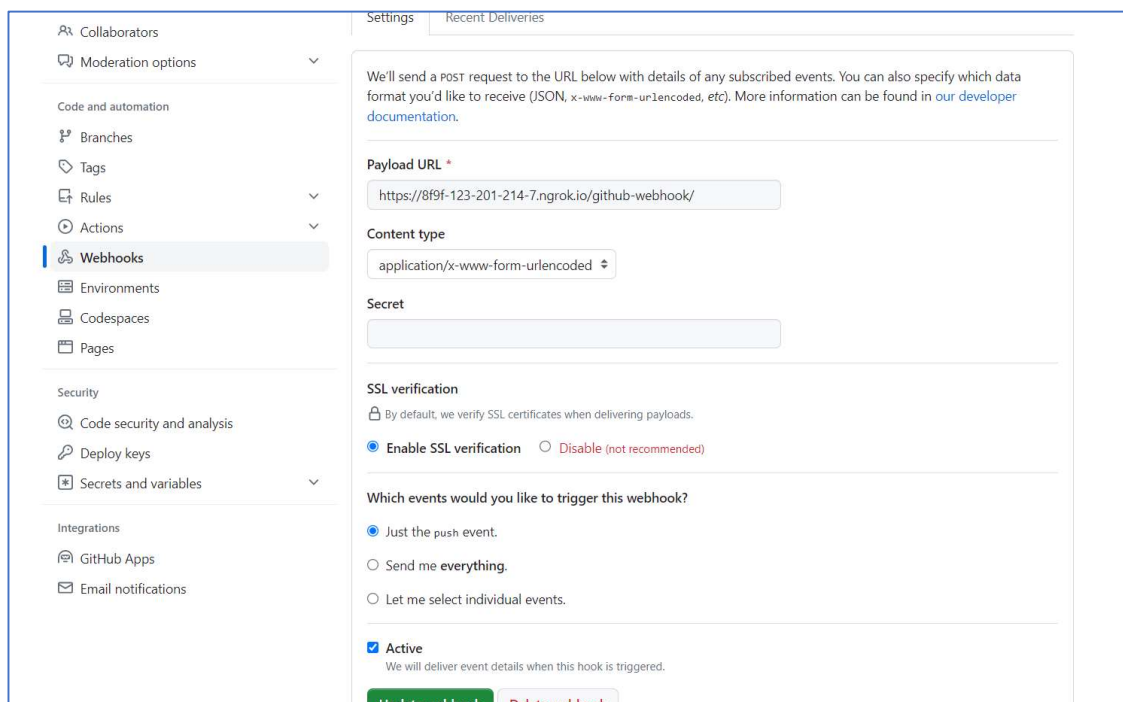
2. In Payload URL we need to enter a URL used send a POST request to with details of any subscribed events

`http://{jenkins-url}/github-webhook`

3. For this we will use ngrok -> download ngrok -> double click on ngrok.exe

4. Enter command `ngrok.exe http 8080` [Jenkins url]

5. Save Web Hook [You can select the trigger events]



6. Edit a file in Git Hub repo

7. Commit changes

This will trigger the build in Jenkins

8. Check Console output

9. Check GitHub Hook log

Webhooks

Webhooks allow notifications to be sent to external web services when certain events (like push/pull/merge) happen on GitHub

We can configure webhooks to trigger events when

- Code is pushed to a repository
- A pull request is opened
- A GitHub Pages site is built

When a webhook is configured, we can select the events we want to subscribe to. To limit the number of http requests to our server we should ideally configure a webhook for a specific event.

By default, webhooks installed on an organization or a repository are only subscribed to the push event.

A server is needed to receive webhooks events. When a webhook is added, we need to specify an url where we want to receive the webhook events. GitHub will send a HTTP POST payload to this URL when any events that the webhook is subscribed to occur.

Repository webhooks allow you to receive HTTP POST payloads whenever certain events happen in a repository.

For GitHub to send webhook payloads, a server should be accessible on internet. For this we use ngrok. Ngrok makes your locally-hosted web server appear to be hosted on a subdomain of ngrok.com