


Setting Up Jenkins Pipeline for a Spring Boot App

Learn to install and configure Jenkins on the local machine. Also, learn to setup a Jenkins pipeline and configure job steps for a Spring boot application including executing the unit tests and archiving the old builds. 1. Creating a Spring Boot Application Spring Boot is a java-based framework ...

 Bayvao Verma

 July 9, 2023

 DevOps

 Ci/CD, Jenkins

Learn to install and configure Jenkins on the local machine. Also, learn to setup a Jenkins pipeline and configure job steps for a Spring boot application including executing the unit tests and archiving the old builds.

1. Creating a Spring Boot Application

Spring Boot is a java-based framework developed by the Pivotal team to create stand-alone, production-grade application. Spring boot is also used to develop applications based on microservices architecture where multiple services run independently and are loosely coupled with each other.

In this tutorial, we are using the application developed for [Spring boot hello world example](#).

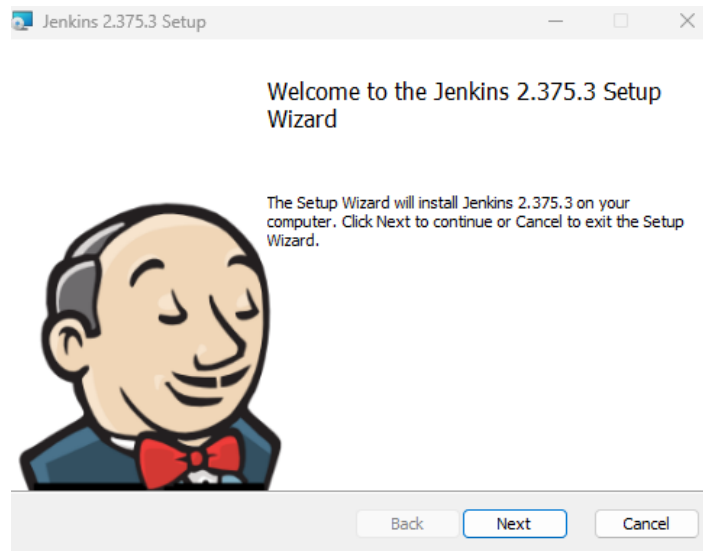
2. Setting Up Jenkins

2.1. Installing Jenkins

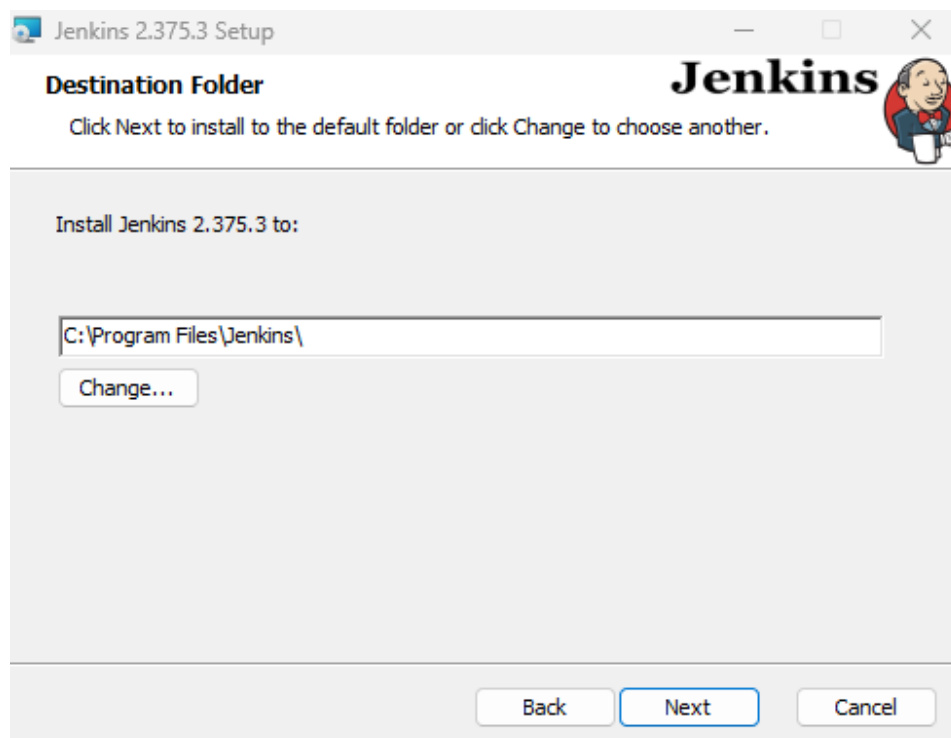
We can go to the Jenkins download link: <https://www.jenkins.io/download/> and download the package suitable for your operating system.

Once the package is downloaded, double-click on it to start installing and follow the below steps:

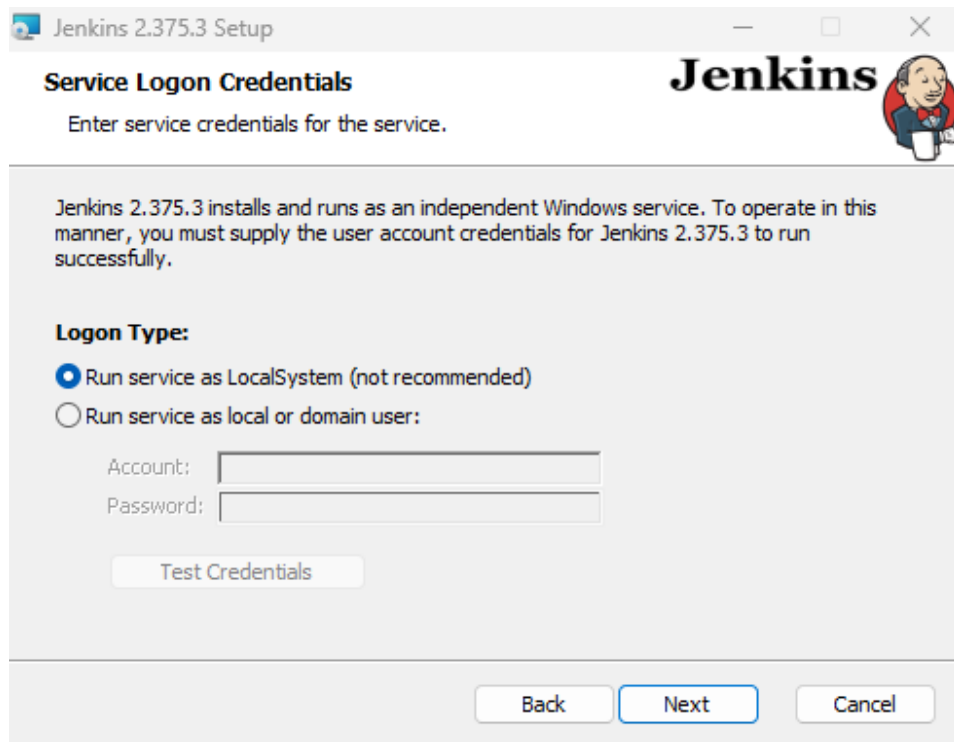
- Double-click the installer and click *Next*.



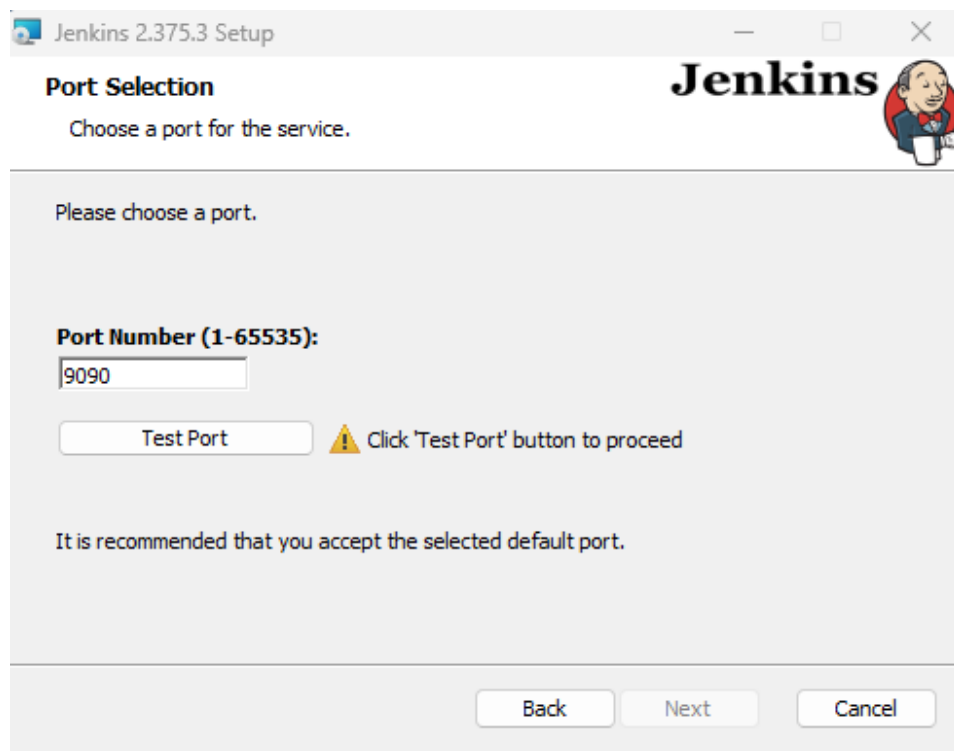
- Choose the directory where we want to install Jenkins and click *Next*.



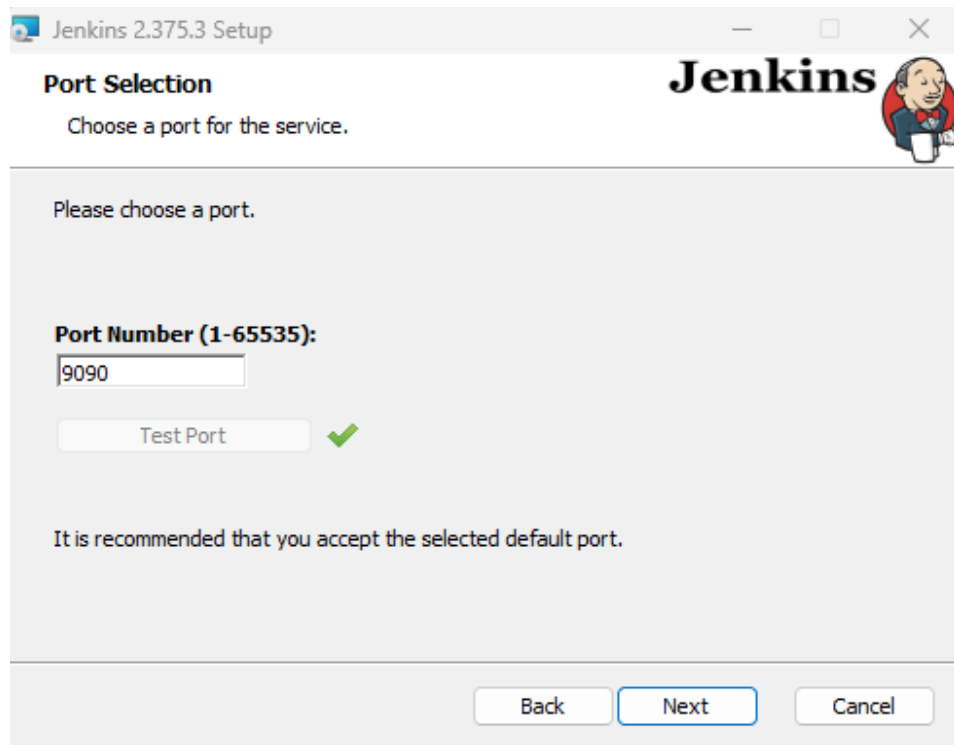
- Next, select the "Logon Type" as "Run service as LocalSystem" which will grant Jenkins full access to our machine and services. Click on *Next* to continue.



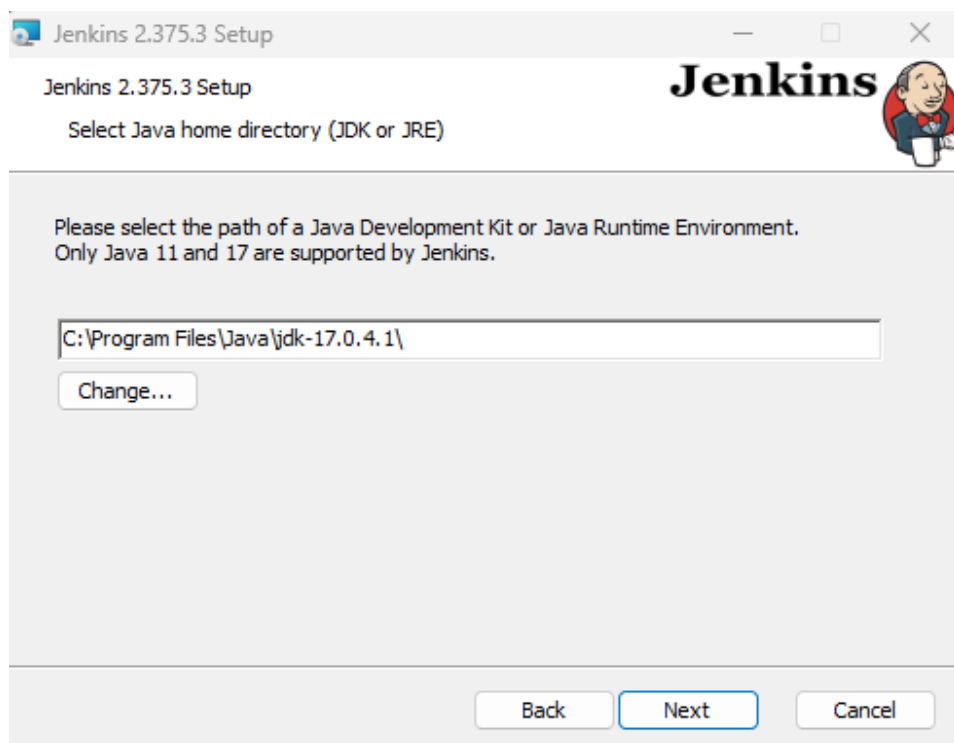
- The next step is port selection. We can specify any HTTP port and click on Test Port to check if the port is available or not. We have used port 9090.



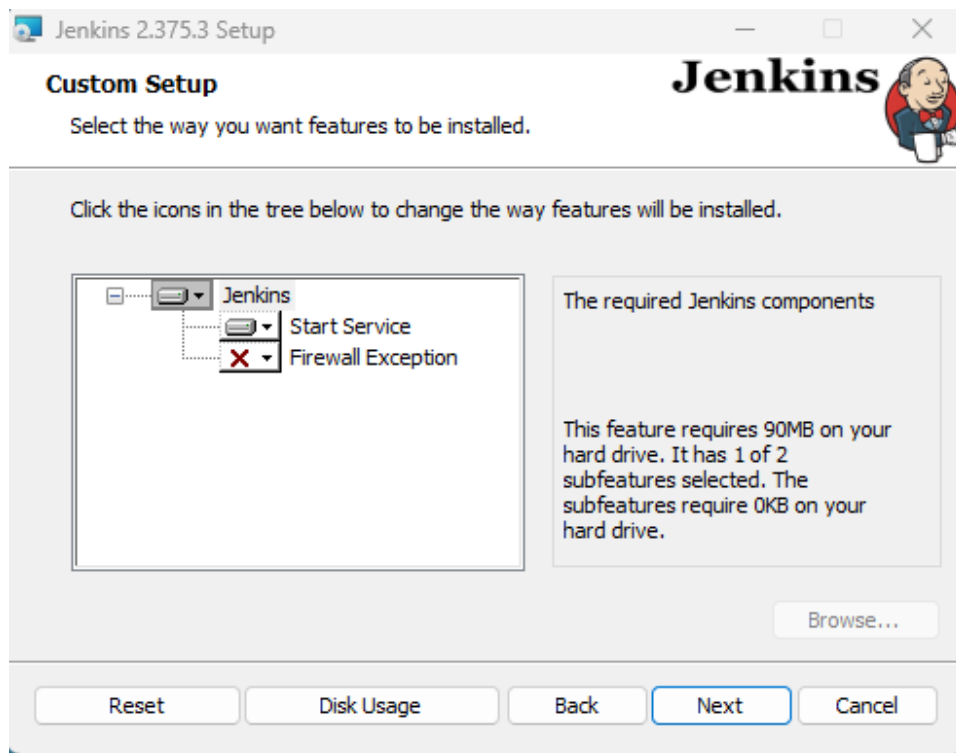
If the mentioned port is available, then we will see a "Green Tick" beside the "Test Port" button like below. Click on *Next* to continue.



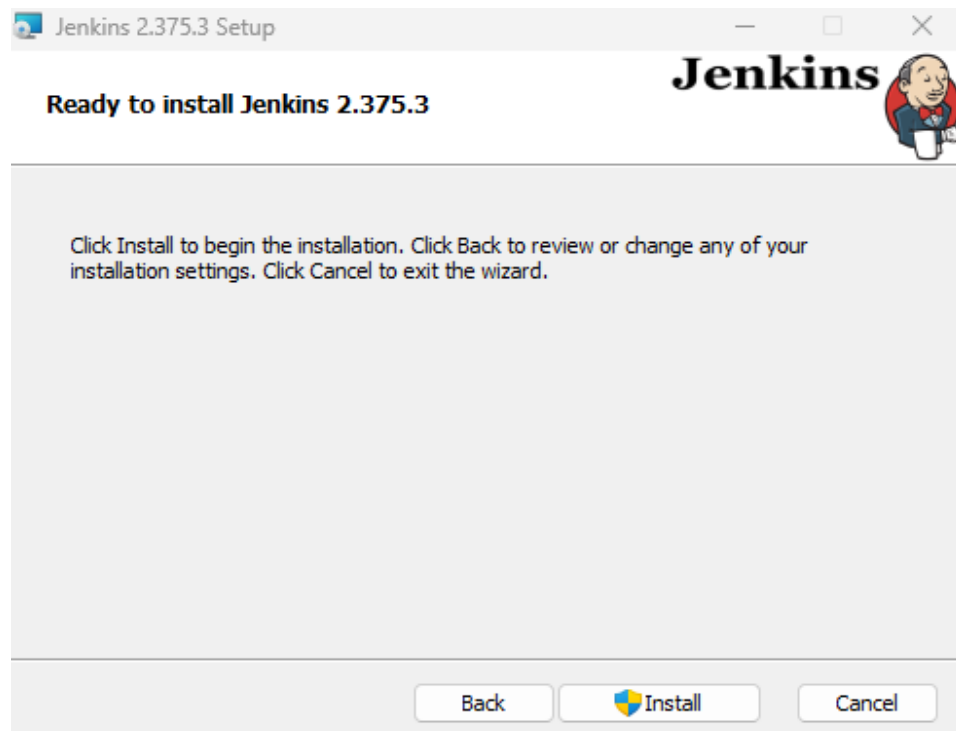
- If we have downloaded the latest version of Jenkins, we need to have java 11 or 17 installed. If we don't have the required version, we need to install it else we have to select the path of the JDK in the system and click *Next*.



- We don't have to do anything in this step; simply click Next to continue. We are testing locally, so firewall selection can be omitted.



- Finally, click on *Install* to complete the installation process. It may take some time to complete the installation.



- Once the installation is completed, we will see this page; now we click on *Finish and* are done with the installation.



2.2 Configuring Jenkins

Once Jenkins is installed, Let's open a browser and go to <http://localhost:9090/> or whatever port we used while installing. We will see a web page something like this –

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

`C:\ProgramData\Jenkins\.jenkins\secrets\initialAdminPassword`

Please copy the password from either location and paste it below.

Administrator password



Continue

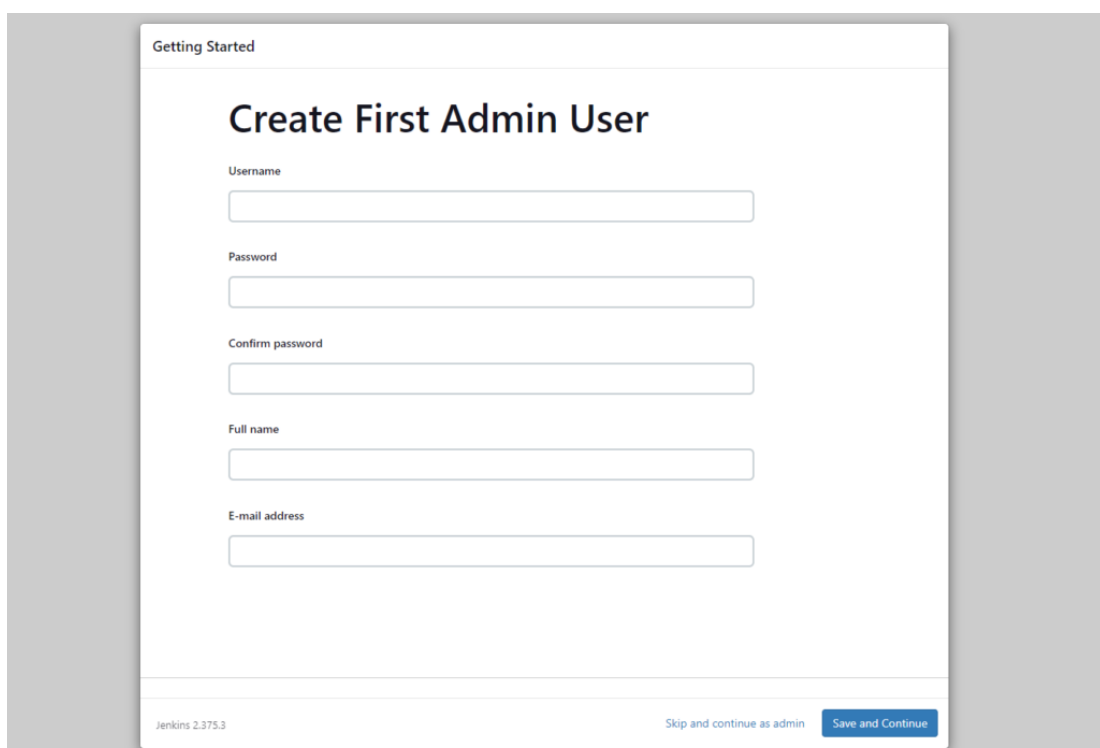
On this page, we need to specify a temporary administrative password, to get the password go to the path mentioned in the browser. In my case, it is, *C:\ProgramData\Jenkins\.jenkins\secrets\initialAdminPassword*

Copy and paste the password in the password field and click on *Continue*. On the next page, we will see something like this-



Here we click on "Install suggested plugins" to let Jenkins download and configure some necessary plugins for us. We wait till the download is completed, this may take a while.

Once the download is completed, we click on *Continue*. The next step is where we need to create an admin user, which we will use while login into Jenkins.

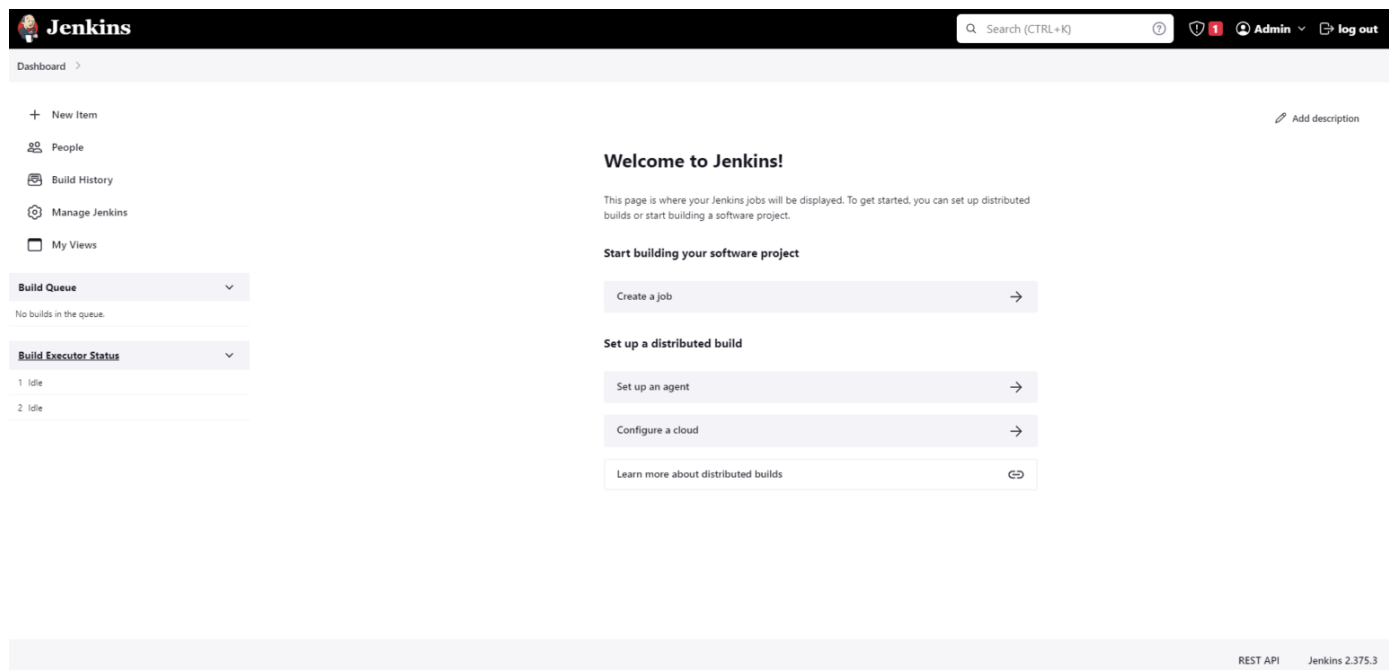
The image shows the 'Getting Started' window of Jenkins with the 'Create First Admin User' form. The form has five input fields: 'Username', 'Password', 'Confirm password', 'Full name', and 'E-mail address'. At the bottom right, there are two buttons: 'Skip and continue as admin' and 'Save and Continue'. The version 'Jenkins 2.375.3' is displayed at the bottom left.

Let's fill out the form, and remember the username and password, as these will be our admin credentials, and click *Save and Continue*.

In the next step, we will be asked if you want to change the URL, but don't change anything click *Continue* and then click *Finish* to complete the Jenkins configuration.

2.3. Install Maven Plugin

Once we are done with the initial Jenkins configuration, we should land on the Jenkins home page –



Click on the "Manage Jenkins" option on the left side of the screen. Then we should see something like this on your screen –

Jenkins

Dashboard > Manage Jenkins

+ New Item

People

Build History

Manage Jenkins

My Views

Build Queue

No builds in the queue.

Build Executor Status

1 Idle

2 Idle

Manage Jenkins

Building on the built-in node can be a security issue. You should set up distributed builds. See [the documentation](#).

Set up agent Set up cloud Dismiss

System Configuration

- Configure System: Configure global settings and paths.
- Global Tool Configuration: Configure tools, their locations and automatic installers.
- Manage Plugins: Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
- Manage Nodes and Clouds: Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

Security

- Configure Global Security: Secure Jenkins; define who is allowed to access/use the system.
- Manage Credentials: Configure credentials
- Configure Credential Providers: Configure the credential providers and types
- Manage Users: Create/delete/modify users that can log in to this Jenkins.

Status Information

- System Information: Displays various environmental information to assist trouble-shooting.
- System Log: System log captures output from `java.util.logging` output related to Jenkins.
- Load Statistics: Check your resource utilization and see if you need more computers for your builds.
- About Jenkins: See the version and license information.

Troubleshooting

Next, we click on the “Manage Plugins” option to install/remove/view the plugins.

Under Manage plugins we click on “Available Plugins” and search for the “Maven Integration” plugin. We selected the option shown in the screenshot below. Then we click on the “Install without restart” button.

Jenkins

Dashboard > Manage Jenkins > Plugin Manager

Updates

Available plugins

Installed plugins

Advanced settings

Download progress

Plugins

Maven Integration

Install	Name	Released
<input checked="" type="checkbox"/>	Maven Integration 3.20 Build Tools This plugin provides a deep integration between Jenkins and Maven. It adds support for automatic triggers between projects depending on SNAPSHOTS as well as the automated configuration of various Jenkins publishers such as Junit.	4 mo 12 days ago
<input type="checkbox"/>	Pipeline Maven Integration 1257.v89e586d3c58c pipeline Maven This plugin provides integration with Pipeline, configures maven environment to use within a pipeline job by calling <code>sh mvn</code> or <code>bat mvn</code> . The selected maven installation will be configured and prepended to the path.	1 mo 20 days ago

Install without restart Download now and install after restart Update information obtained: 1 day 22 hr ago Check now

There were errors checking the update sites: `UnknownHostException: updates.jenkins.io`

Once the downloading starts we will see something like this –

Dashboard > Manage Jenkins > Plugin Manager

UKHttp	✓ Success
GitHub API	✓ Success
Git	✓ Success
GitHub	✓ Success
GitHub Branch Source	✓ Success
Pipeline: GitHub Groovy Libraries	✓ Success
Pipeline Graph Analysis	✓ Success
Pipeline: REST API	✓ Success
Pipeline: Stage View	✓ Success
Git	✓ Success
SSH Build Agents	✓ Success
Matrix Authorization Strategy	✓ Success
PAM Authentication	✓ Success
LDAP	✓ Success
Email Extension	✓ Success
Mailer	✓ Success
Loading plugin extensions	✓ Success
Gradle	✓ Success
Loading plugin extensions	✓ Success
Javadoc	✓ Success
JSch dependency	✓ Success
Maven Integration	✓ Success
Loading plugin extensions	✓ Success

[Go back to the top page](#)
(you can start using the installed plugins right away)

☐ Restart Jenkins when installation is complete and no jobs are running

REST API Jenkins 2.375.3

We will wait for the processes to complete. Optionally if we want we can select the option "Restart Jenkins when installation is complete and no jobs are running".

Now go to *Jenkins Dashboard* → *Manage Jenkins* → *Global Tool Configuration*.

Under this scroll down and we find a section for *Maven*. We click on the Add Maven button. Under Name, we mention a *unique name*, uncheck "Install automatically" and in the [MAVEN_HOME](#) section mention the path to the location of maven installed in our system, and click on *Save*.

We can also add multiple versions of maven by specifying a unique name for each one of them.

Maven

Maven installations

List of Maven installations on this system

Add Maven

Maven
Name

jenkins-maven

MAVEN_HOME

C:\Program Files\Apache Software Foundation\apache-maven-3.8.1

☐ Install automatically ?

Add Maven

2.4. Installing the Git Plugin

The next step involves installing Git plugin. Jenkins installs the plugin in the initial configuration. From the *Jenkins Dashboard*, go to *Manage Jenkins* → *Manage Plugins* → *Installed Plugins* and search for "GIT", we should have a bunch of git plugins already installed, If it is not available, then we need to install it the way we installed the Maven Plugin.

Dashboard > Manage Jenkins > Plugin Manager

Updates

Available plugins

Installed plugins

Advanced settings

Plugins

Q GIT /

Name	Enabled
<div><div>Git client plugin 4.1.0</div><div>Utility plugin for Git support in Jenkins</div><div>Report an issue with this plugin</div></div>	<div><div></div><div></div></div>
<div><div>Git plugin 5.0.0</div><div>This plugin integrates Git with Jenkins.</div><div>Report an issue with this plugin</div></div>	<div><div></div><div></div></div>
<div><div>GitHub API Plugin 1.303-400.v35c2d8258028</div><div>This plugin provides GitHub API for other plugins.</div><div>Report an issue with this plugin</div></div>	<div><div></div><div></div></div>
<div><div>GitHub Branch Source Plugin 1701.v00cc8184df93</div><div>Multibranch projects and organization folders from GitHub. Maintained by CloudBees, Inc.</div><div>Report an issue with this plugin</div></div>	<div><div></div><div></div></div>
<div><div>GitHub plugin 1.37.0</div><div>This plugin integrates GitHub to Jenkins.</div><div>Report an issue with this plugin</div></div>	<div><div></div><div></div></div>
<div><div>Pipeline: GitHub Groovy Libraries 38.v445716ea_eddda_</div><div>Allows Pipeline Groovy libraries to be loaded on the fly from GitHub.</div><div>Report an issue with this plugin</div></div>	<div><div></div><div></div></div>


REST API Jenkins 2.375.3

3. Building a Maven Project


In this section, we will learn how to create a job, clone a git repository and build a jar using Jenkins.


3.1. Creating a Jenkins job


To create a Jenkins job, log in to Jenkins and go to Jenkins Dashboard, then click on “New Item” to create a new job.


 **Jenkins**


Dashboard >

 New Item

 People

 Build History

 Manage Jenkins

 My Views

Build Queue

No builds in the queue.


Build Executor Status


1 Idle


2 Idle


Once we click on “New Item”, you should see a page like this –


» Required field


**Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.


**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**
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**Multibranch Pipeline**
Creates a set of Pipeline projects according to detected branches in one SCM repository.

**Organization Folder**
Creates a set of multibranch project subfolders by scanning for repositories.

On this page, we enter the name of our job. It can be anything, but the best practice is to enter the job name the same as the project/repository name for which we are building the job.

After entering the job name, select “Freestyle project” as we want to have a granular controller over everything that we configure.

Next click *OK* to continue.

3.2. Cloning Git/GitHub Repository in Jenkins

Once the job configuration is successful, we will see a page like this –

Dashboard > spring-jenkins-demo > Configuration

General Enabled

Configure

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

General

Description

[Plain text] [Preview](#)

☐ Discard old builds ?

☐ GitHub project

☐ This project is parameterized ?

☐ Throttle builds ?

☐ Execute concurrent builds if necessary ?

[Advanced...](#)

Source Code Management

☒ None

☐ Git ?

Build Triggers

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

[Save](#) [Apply](#)

On this page, click on “Source Code Management” (SCM), a tool used by developers/ organizations to manage application code. E.g. GitHub, GitLab. Under Source Code Management, we select Git and enter the Repository URL.

Under the “Branches to Build” section, we specify the default branch for which we want to trigger the pipeline. If our repository is a public repository, then we don’t need to set any credentials, else we need to mention our GitHub credentials.

Dashboard > spring-jenkins-demo > Configuration

Configure

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

Source Code Management

☐ None

☒ Git ?

Repositories ?

Repository URL ?

[https://github.com/Bayvao/spring-jenkins-demo.git](#)

Credentials ?

- none -

[+ Add](#)

[Advanced...](#)

[Add Repository](#)

Branches to build ?

Branch Specifier (blank for 'any') ?

*/master

[Add Branch](#)

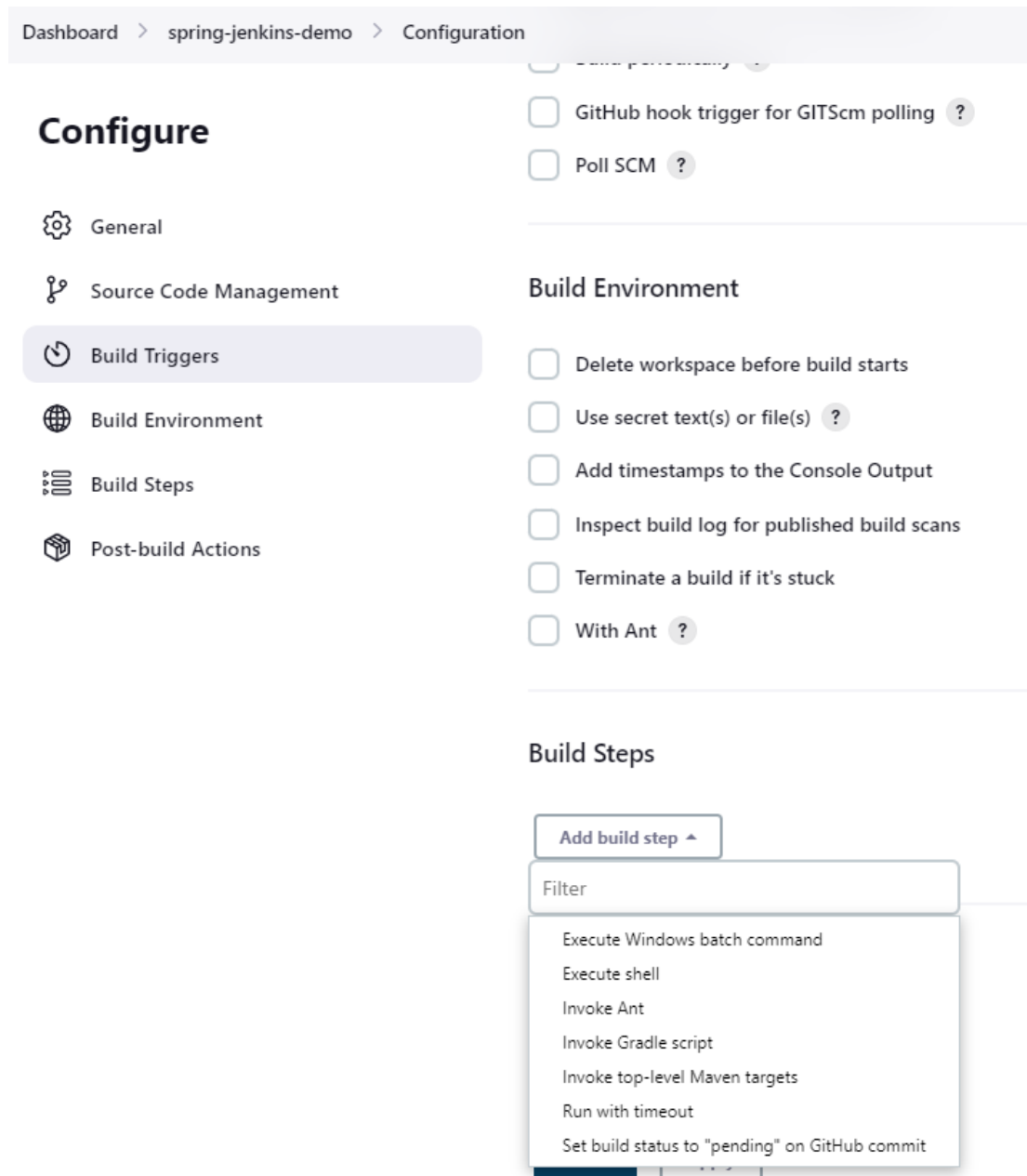
[Save](#) [Apply](#)

Finally, click on *Save*.

3.3. Build a Jar file

Now under our Jenkins job, we click on configure to continue further configuration.

We go to *Build Steps* section, click on *Add Build Steps*, and select "Invoke top-level Maven targets".



Now under Maven Version, we select the one which we created earlier, i.e. *jenkins-maven*.

Under Goals, we write maven command **-B -DskipTests clean package**. Finally, click on *Save*.

Build Steps

≡

Invoke top-level Maven targets ?

Maven Version

jenkins-maven

Goals

-B -DskipTests clean package

Advanced...

Add build step ▼

Post-build Actions

Add post-build action ▼

Save

Apply

3.4. Executing Unit Tests

We repeat the same steps again, we go to *Build Steps*, click on *Add Build Steps* to add another step, and select "Invoke top-level Maven targets".

Now under Maven Version, we select the one which we created earlier, but now in this Goal write "test", and click on *Save* to continue.

Build Steps

≡ Invoke top-level Maven targets ?

Maven Version

jenkins-maven

Goals

-B -DskipTests clean package

Advanced...

≡ Invoke top-level Maven targets ?

Maven Version

jenkins-maven

Goals

test

Advanced...

Add build step ▾


Save

Apply

3.5. Executing the Job

Once we complete the above configuration, we go to our job in Jenkins job, and click on Build Now.

Dashboard > maven-jenkins-demo >

 Status

 Changes

 Workspace

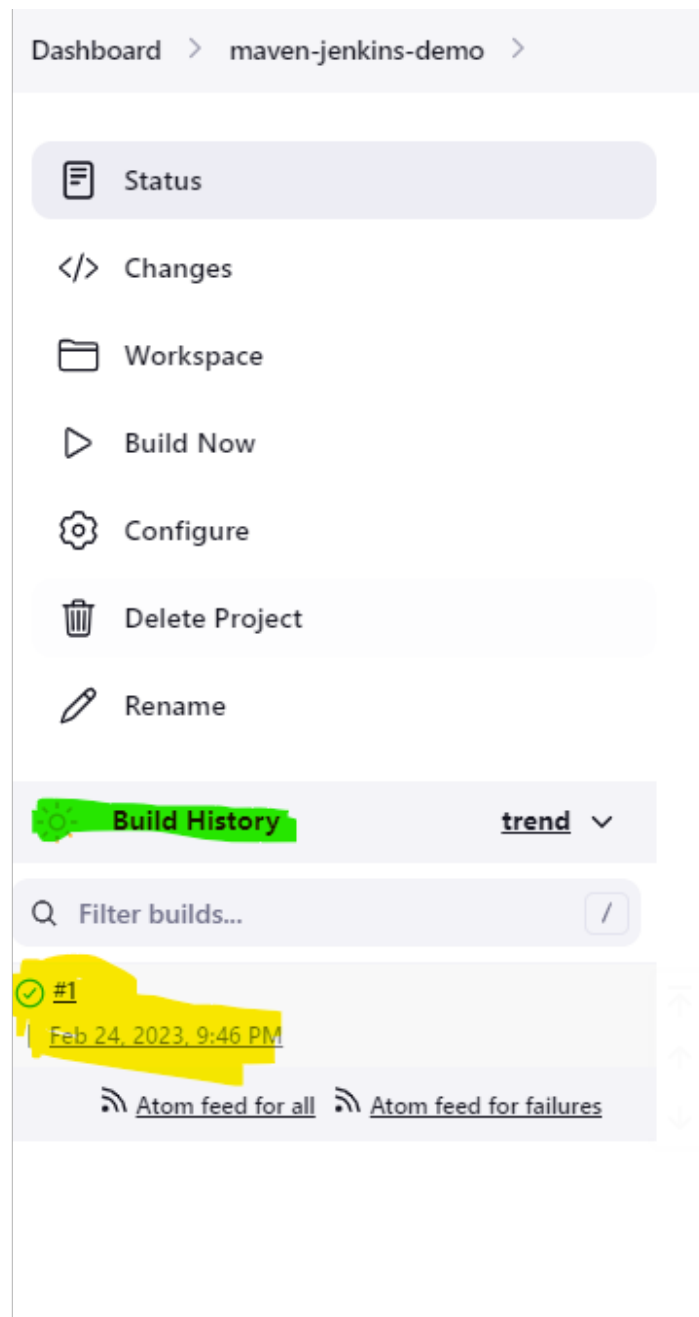
 Build Now

 Configure

 Delete Project

 Rename


Once we click on Build Now, our build will start, which we can check in the “Build History” section as shown below.





If our build is a success, we will see a **green** checkbox, beside our build number, else we will see a **red** cross mark if the build fails.


3.6. Verify Build Logs


To check logs of the build, we just click on the build number, and we will see a page like this –


 Status


 Changes


 Console Output


 Edit Build Information


 Delete build '#1'

 Git Build Data

 **Build #1 (Feb 24, 2023, 9:46:59 PM)**

 No changes.


 Started by user [Admin](#)


 **Revision:** adb7907f50ddd476ab13442865b9172e626cdb4f
Repository: <https://github.com/Bayvao/maven-jenkins-demo.git>


- refs/remotes/origin/master


Now we click on “Console Output”, and we will see all the logs generated for this build.


 Status


 Changes


 Console Output

 View as plain text

 Edit Build Information

 Delete build '#1'

 Git Build Data

 **Console Output**

```
Started by user Admin
Running as SYSTEM
Building in workspace C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo
The recommended git tool is: NONE
No credentials specified
> git.exe rev-parse --resolve-git-dir C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\.git # timeout=10
Fetching changes from the remote Git repository
> git.exe config remote.origin.url https://github.com/Bayvao/maven-jenkins-demo.git # timeout=10
Fetching upstream changes from https://github.com/Bayvao/maven-jenkins-demo.git
> git.exe --version # timeout=10
> git --version # 'git version 2.28.0.windows.1'
> git.exe fetch --tags --force --progress -- https://github.com/Bayvao/maven-jenkins-demo.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git.exe rev-parse "refs/remotes/origin/master" # timeout=10
Checking out Revision adb7907f50ddd476ab13442865b9172e626cdb4f (refs/remotes/origin/master)
> git.exe config core.sparsecheckout # timeout=10
> git.exe checkout -f adb7907f50ddd476ab13442865b9172e626cdb4f # timeout=10
Commit message: "Update README.md"
First time build. Skipping changelog.
[maven-jenkins-demo] $ cmd.exe /C "C:\ProgramData\Jenkins\jenkins\tools\hudson.tasks.Maven\MavenInstallation\jenkins-maven\bin\mvn.cmd -B -DskipTests clean package && exit %ERRORLEVEL%"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.mycompany.app:maven-jenkins-demo -----
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- clean:3.2.0:clean (default-clean) @ maven-jenkins-demo ---
[INFO] Deleting C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\target
[INFO]
```

Now if we look closely at the logs, we will see after the build is a success, a jar file is created and stored in your system. *Copy this path*

```

[INFO] skip non existing resourceDirectory C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\src\main\resources
[INFO] --- compiler:3.10.1:compile (default-compile) @ maven-jenkins-demo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\target\classes
[INFO] --- resources:3.3.0:testResources (default-testResources) @ maven-jenkins-demo ---
[INFO] skip non existing resourceDirectory C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\src\test\resources
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ maven-jenkins-demo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\target\test-classes
[INFO] --- surefire:3.0.0-M8:test (default-test) @ maven-jenkins-demo ---
[INFO] Tests are skipped.
[INFO] --- jar:3.3.0:jar (default-jar) @ maven-jenkins-demo ---
[INFO] Building jar: C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\target\maven-jenkins-demo-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.434 s
[INFO] Finished at: 2023-02-24T21:47:10+05:30
[INFO] -----
[maven-jenkins-demo] $ cmd.exe /C "C:\ProgramData\Jenkins\jenkins\tools\hudson.tasks.Maven\MavenInstallation\jenkins-maven\bin\mvn.cmd test && exit %ERRORLEVEL%"
[INFO] Scanning for projects...
[INFO] -----< com.mycompany.app:maven-jenkins-demo >-----
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO] --- resources:3.3.0:resources (default-resources) @ maven-jenkins-demo ---
[INFO] skip non existing resourceDirectory C:\ProgramData\Jenkins\jenkins\workspace\maven-jenkins-demo\src\main\resources
[INFO]

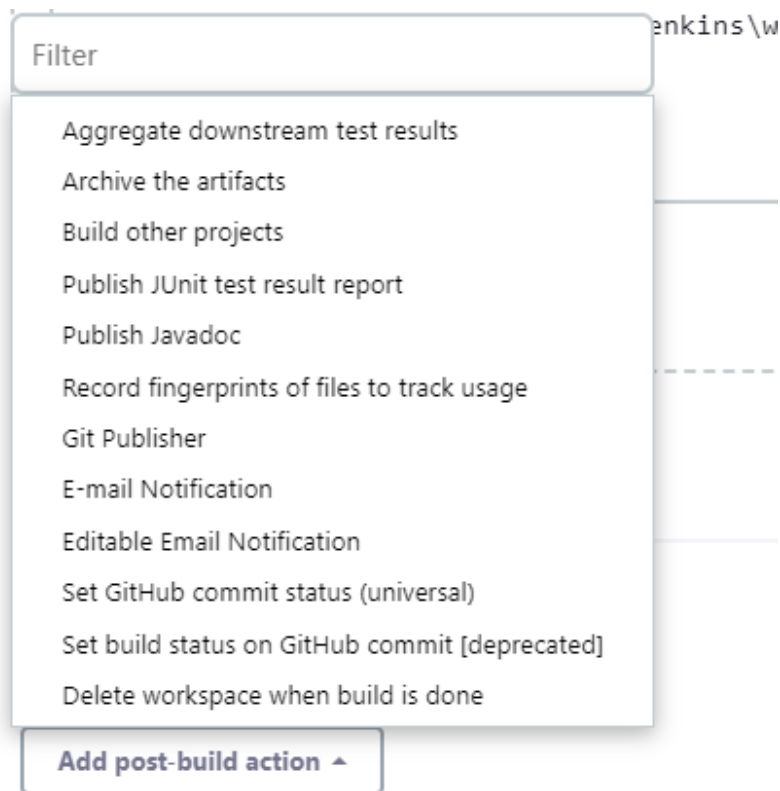
```

3.7. Archiving the Artifact

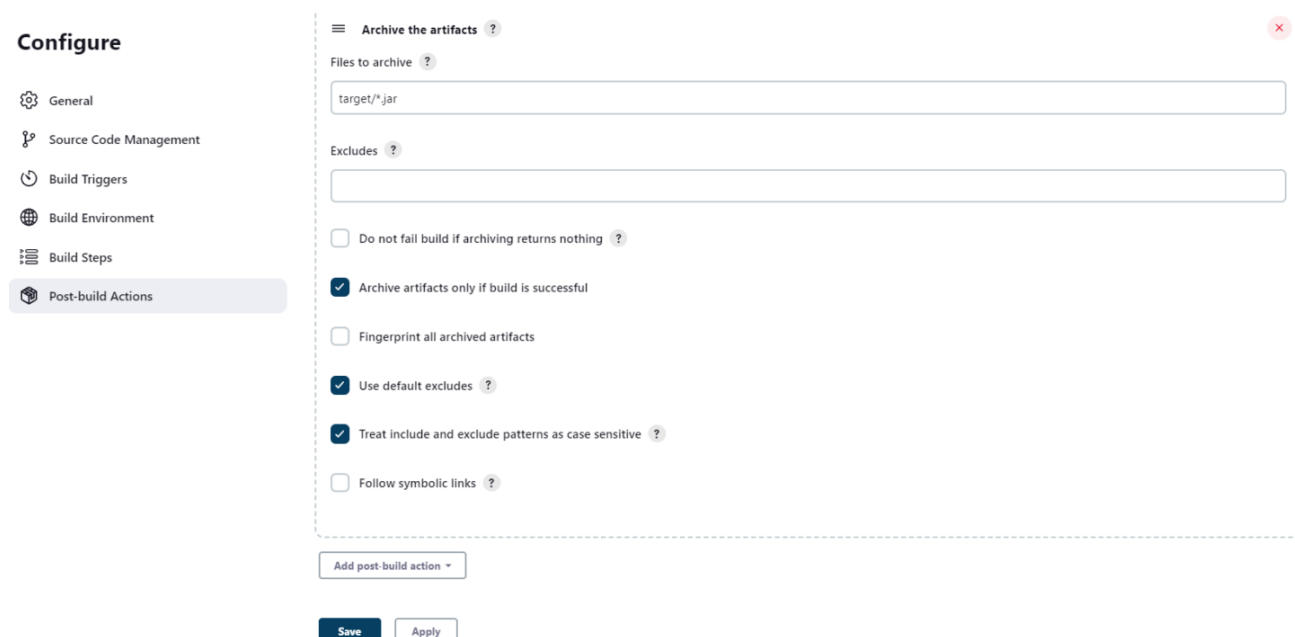
In DevOps, the output of the build i.e. jar file/war file etc., is called the artifact and it needs to be archived because once the artifact is built and deployed, it has no use and is consuming memory in our system, so after a successful build and deployment we have to archive the jar from the local memory.

To archive the artifact, we would be adding "Post-build Actions". So inside our Jenkins job, we go to configure and add a Post-build Action.



A drop-down will appear, there we will select "Archive the artifacts".



Once done, specify the name of the file to archive i.e the jar file, we can use wildcards to avoid specifying the exact name, like – *target/*.jar*. Then we click on advanced and select the checkbox for “Archive artifacts only if the build is successful”, as the name mentions we archive once a build is success and deployed successfully, like below –



Click on *Save*. and hit *Build Now*, Once the build is successful, we will see the last success jar file displayed on the home page of our job.

 Status Changes Workspace Build Now Configure Delete Project Rename Build Historytrend ▼ Filter builds...  #3| [Feb 24, 2023, 10:55 PM](#) #2| [Feb 24, 2023, 10:08 PM](#) #1| [Feb 24, 2023, 9:46 PM](#) [Atom feed for all](#)  [Atom feed for failures](#)

Project maven-jenkins-demo



Last Successful Artifacts

 [maven-jenkins-demo-1.0-SNAPSHOT.jar](#) 2.64 KB [view](#)

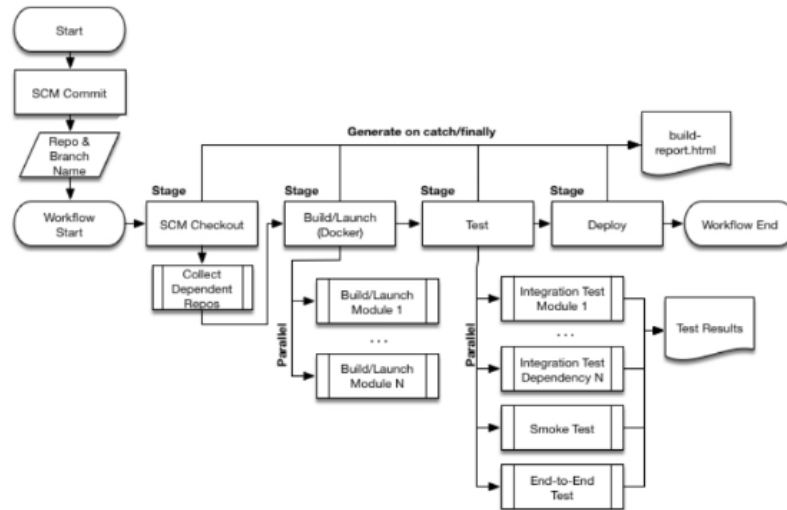
Permalinks

- [Last build \(#3\), 41 sec ago](#)
- [Last stable build \(#3\), 41 sec ago](#)
- [Last successful build \(#3\), 41 sec ago](#)
- [Last completed build \(#3\), 41 sec ago](#)

4. Jenkins Pipeline and Jenkinsfile

4.1. Jenkins Pipeline

A pipeline is a collection of steps/stages executed in the Continuous Integration and Continuous Deployment (CI/CD) process. A pipeline step consists of checkout, build, test, and deployment of your code. A Sample pipeline diagram is below.



4.2. Jenkinsfile

A Jenkinsfile is a text file where we define our pipeline as a code. The Jenkinsfile exists within our project repository.

There are 2 types of JenkinsFile:

- Declarative pipeline
- Scripted Pipeline

We are going to use the **Declarative pipeline** in our tutorial. An example of such a Declarative pipeline JenkinsFile with 3 stages is shown below:

- Build
- Test
- Deploy

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
    }
}
```



```

    }
    stage('Deploy') {
        steps {
            //
        }
    }
}

```

- **pipeline** – keyword, with which every Jenkinsfile must start with.
- **agent** – this means using which Jenkins environment will the entire pipeline or a particular stage will run, examples of agents can be maven, docker, etc. The agent any means, run the pipeline on any available agent.
- **stages** – consists of one more stage.
- **stage** – a task of our workflow.
- **steps** – It is where we define what to do as part of the task.

4.3. Installing the Jenkins Pipeline Plugin

If we have installed Jenkins suggested plugin during the installation process, then pipeline plugins should already be installed for us by Jenkins.

To verify this, go to Dashboard → Manage Jenkins → Manage Plugins → Installed Plugins, and search for “pipeline”, we should see them here.

Updates

Available plugins

Installed plugins

Advanced settings

Plugins


Name 1	Enabled
Pipeline 590.v6a_d052e5a_a_b_5 A suite of plugins that lets you orchestrate automation, simple or complex. See Pipeline as Code with Jenkins for more details. Report an issue with this plugin	<input checked="" type="checkbox"/> <input type="checkbox"/>
Pipeline Graph Analysis Plugin 202.va_d268e64deb_3 Provides a REST API to access pipeline and pipeline run data. Report an issue with this plugin	<input checked="" type="checkbox"/> <input type="checkbox"/>
Pipeline: API 1208.v0cc7c6e0da_9e Plugin that defines Pipeline API. Report an issue with this plugin	<input checked="" type="checkbox"/> <input type="checkbox"/>
Pipeline: Basic Steps 1010.vf7a_b_98e847c1 Commonly used steps for Pipelines. Report an issue with this plugin	<input checked="" type="checkbox"/> <input type="checkbox"/>
Pipeline: Build Step 2.18.1 Adds the Pipeline step <code>build</code> to trigger builds of other jobs. Report an issue with this plugin	<input checked="" type="checkbox"/> <input type="checkbox"/>
Pipeline: Declarative 2.2118.v31fd5b_9944b_5 An opinionated, declarative Pipeline. Report an issue with this plugin	<input checked="" type="checkbox"/> <input type="checkbox"/>
Pipeline: Declarative Extension Points API 2.2118.v31fd5b_9944b_5 APIs for extension points used in Declarative Pipelines.	<input checked="" type="checkbox"/> <input type="checkbox"/>

4.4. Building a Simple Pipeline


To create a pipeline project, we go to Dashboard → New Item, enter a project name, and select the type as a pipeline. Then we Click *OK* to continue.

Enter an item name


» Required field

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**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**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

**Organization Folder**

Creates a set of multibranch project subfolders by scanning for repositories.

Now on the project configuration page, we go to the Pipeline section. We select the Definition as “pipeline script” and copy-paste the below script –

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo "Building"
      }
    }
    stage('Test') {
      steps {
```



```

        echo "Testing"
    }
}
stage('Deploy') {
    steps {
        echo "Deploying"
    }
}
}
}

```

Here we have written a simple pipeline script, which will just print a string in the console during the execution of each stage.

Configure



General



Advanced Project Options



Pipeline

Pipeline

Definition

Pipeline script

Script ?

```

1 pipeline {
2   agent any
3   stages {
4     stage('Build') {
5       steps {
6         echo "Building"
7       }
8     }
9     stage('Test') {
10      steps {
11        echo "Testing"
12      }
13    }
14    stage('Deploy') {
15      steps {
16        echo "Deploying"
17      }
18    }
19  }
20 }

```

☒ Use Groovy Sandbox ?

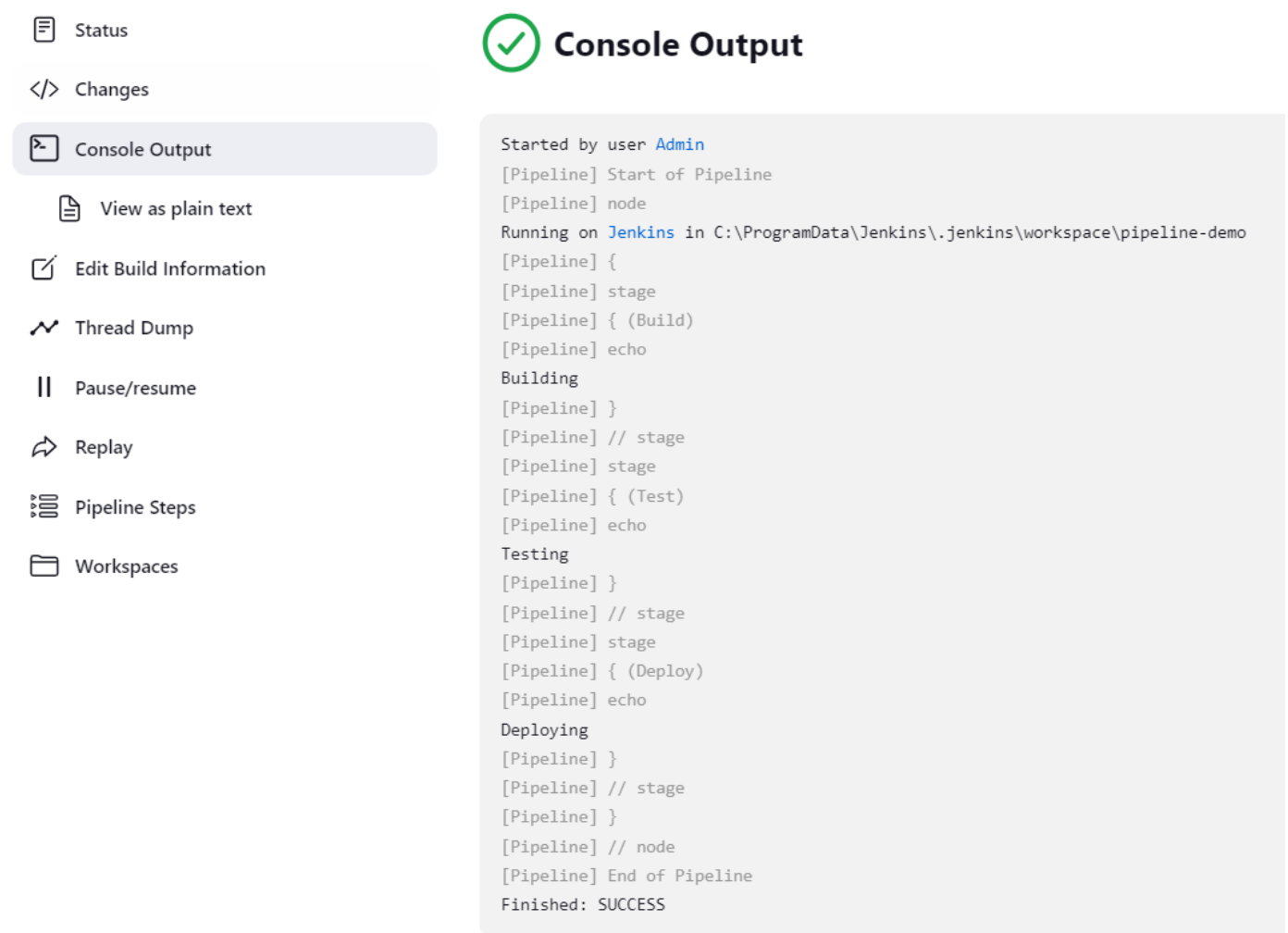
[Pipeline Syntax](#)

Save

Apply

Click on *Save*.

Next we click on “Build Now” to run the pipeline. Then check the console output, and we would see the pipeline stages running sequentially and printing the values that we specified.



The screenshot displays the Jenkins interface for a pipeline run. On the left, a sidebar contains navigation links: Status, Changes, Console Output (highlighted), View as plain text, Edit Build Information, Thread Dump, Pause/resume, Replay, Pipeline Steps, and Workspaces. The main area is titled 'Console Output' with a green checkmark icon. It shows the following log output:

```
Started by user Admin
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in C:\ProgramData\Jenkins\.jenkins\workspace\pipeline-demo
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
Building
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
Testing
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
Deploying
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

4.5. Adding Steps to Jenkins Pipeline

Each step of our pipeline can consist of more than one task which has to be done, and it may be that we need to execute a batch (bat) script or a shell (sh) script (depending on the system being used) in the pipeline.

To do that, we can just modify the pipeline script like the below –

Pipeline

Definition

Pipeline script

Script ?

```
1 pipeline {  
2   agent any  
3   stages {  
4     stage('Build') {  
5       steps {  
6         bat 'echo "Building"'  
7       }  
8     }  
9     stage('Test') {  
10      steps {  
11        bat 'echo "Testing"'  
12        bat ''  
13        echo "Testing more data"  
14      }  
15    }  
16    stage('Deploy') {  
17      steps {  
18        bat 'echo "Deploying"'  
19      }  
20    }  
21  }  
22 }  
23 }  
24 }
```

☒ Use Groovy Sandbox ?

[Pipeline Syntax](#)

Save

Apply

Click Save. and do Build Now. We can check the console to see the output.

4.6. Retry and Timeouts

Consider some failure that happens while executing a stage of our pipeline and before failing the entire job, we want to retry the steps of stage a certain number of times before we consider it to be a failure. To do that we use the ***retry(n)*** function.

Below code, sample illustrates the use of *retry(n)* –



```
stage('Deploy') {  
  steps {  
    retry(3) {  
      bat 'I am not going to deploy :c'  
    }  
  }  
}
```

Update the pipeline script and Save.

Script ?

```
1 pipeline {  
2   agent any  
3   stages {  
4     stage('Build') {  
5       steps {  
6         bat 'echo "Building"'  
7       }  
8     }  
9     stage('Test') {  
10      steps {  
11        bat 'echo "Testing"'  
12        bat ''  
13        echo "Testing more data"  
14      }  
15    }  
16    stage('Deploy') {  
17      steps {  
18        retry(3) {  
19          bat 'I am not going to deploy :c'  
20        }  
21      }  
22    }  
23  }  
24 }  
25 }  
26 }
```

Now hit “Build Now” and check the console output, we would see the deployment stage executing 3 times before the job is considered a failure.


```
C:\ProgramData\Jenkins\.jenkins\workspace\pipeline-demo>I am not going to deploy :c
'I' is not recognized as an internal or external command,
operable program or batch file.
[Pipeline] }
ERROR: script returned exit code 1
Retrying
[Pipeline] {
[Pipeline] bat
```

```
C:\ProgramData\Jenkins\.jenkins\workspace\pipeline-demo>I am not going to deploy :c
'I' is not recognized as an internal or external command,
operable program or batch file.
[Pipeline] }
ERROR: script returned exit code 1
Retrying
[Pipeline] {
[Pipeline] bat
```

```
C:\ProgramData\Jenkins\.jenkins\workspace\pipeline-demo>I am not going to deploy :c
'I' is not recognized as an internal or external command,
operable program or batch file.
[Pipeline] }
[Pipeline] // retry
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
ERROR: script returned exit code 1
Finished: FAILURE
```

Under how many seconds should a process be completed before considering it a failure? To do that we use the *timeout* (time: *n*, unit: 'SECONDS') function, inside our stage.

Below code, sample illustrates the use of timeout –

```
stage('Deploy') {
    steps {
        retry(3) {
            bat 'echo "Deploying"'
        }
        timeout(time: 3, unit: 'SECONDS') {
            bat 'ping -n 10 127.0.0.1'
        }
    }
}
```



Now let's update our script to as shown below –

Script ?

```
1 pipeline {
2   agent any
3   stages {
4     stage('Build') {
5       steps {
6         bat 'echo "Building"'
7       }
8     }
9     stage('Test') {
10      steps {
11        bat 'echo "Testing"'
12        bat ''
13        echo "Testing more data"
14      }
15    }
16    stage('Deploy') {
17      steps {
18        retry(3) {
19          bat 'echo "Deploying"'
20        }
21        timeout(time: 3, unit: 'SECONDS') {
22          bat 'ping -n 10 127.0.0.1'
23        }
24      }
25    }
26  }
27 }
28
29
```

Click Save and Build Now. Then we check the console output, we would see *ping -n 10 127.0.0.1* process waited for 3 seconds to finish before aborting and failing the pipeline.

```
C:\ProgramData\Jenkins\.jenkins\workspace\pipeline-demo>ping -n 10 127.0.0.1
```

```
Pinging 127.0.0.1 with 32 bytes of data:
```

```
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
```

```
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
```

```
Cancelling nested steps due to timeout
```

```
Sending interrupt signal to process
```

```
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
```

```
Ping statistics for 127.0.0.1:
```

```
    Packets: Sent = 3, Received = 3, Lost = 0 (0% loss),
```

```
Approximate round trip times in milli-seconds:
```

```
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

```
Control-C
```

```
^CTerminate batch job (Y/N)?
```

```
^Cscript returned exit code -1073741510
```

```
[Pipeline] }
```

```
[Pipeline] // timeout
```

```
[Pipeline] }
```

```
[Pipeline] // stage
```

```
[Pipeline] }
```

```
[Pipeline] // node
```

```
[Pipeline] End of Pipeline
```

```
Timeout has been exceeded
```

```
Finished: ABORTED
```

4.7. Environment Variables and Credentials

In a Jenkins pipeline, we write the below command to specify environment variables.

```
environment {  
    NAME = 'John'  
    LASTNAME = 'Doe'  
}
```

To use this we simply specify `$NAME $LASTNAME` (if using `sh` command) or `%NAME% %LASTNAME%` (for `bat` command) inside our pipeline code.

The complete pipeline code would look like the below –

Script ?

```
1 pipeline {  
2  
3     agent any  
4  
5     environment {  
6         NAME = 'John'  
7         LASTNAME = 'Doe'  
8     }  
9  
10    stages {  
11        stage('Build') {  
12            steps {  
13                bat 'echo "My name is %NAME% %LASTNAME%"'  
14            }  
15        }  
16        stage('Test') {  
17            steps {  
18                bat 'echo "Testing"'  
19                bat ''  
20                echo "Testing more data"  
21            }  
22        }  
23        stage('Deploy') {  
24            steps {  
25                retry(3) {  
26                    bat 'echo "Deploying"'  
27                }  
28                timeout(time: 3, unit: 'SECONDS') {  
29                    bat 'ping -n 1 127.0.0.1'  
30                }  
31            }  
32        }  
33    }  
34 }  
35  
36 }
```

We click on Save, hit Build Now, then check the output, we would see the name is printed in

the logs.

```
C:\ProgramData\Jenkins\.jenkins\workspace\pipeline-demo>echo "My name is John Doe"
"My name is John Doe"
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] bat
```

There are times when we need to use secrets inside our pipeline code, in those scenarios we use the **credentials(name)** function.

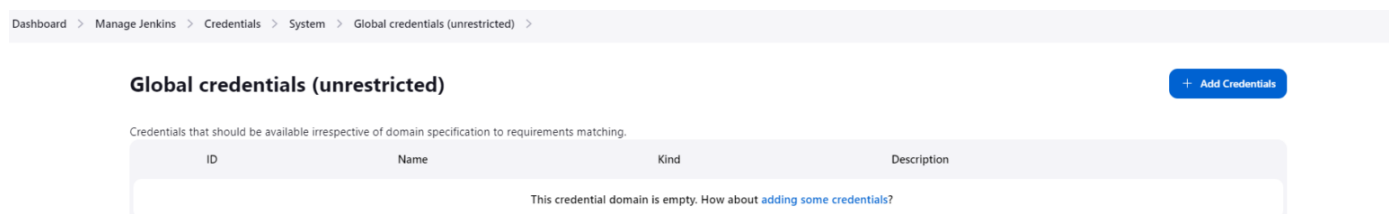
Code sample –

```
environment {
    secretValue = credentials('SECRET_TEXT')
}
```

Basically, this credentials ("") function will fetch credentials whose id is a *SECRET_TEXT* and store it in the variable secretValue.

To create a credential in Jenkins, we follow the below steps –

Dashboard → Manage Jenkins → Manage credentials → system → Global Credentials → Add Credentials



We select kind as the secret text, enter any random value as secret, and we give a unique id for this secret.

New credentials

Kind

Secret text

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Secret

....

ID ?

SECRET_TEXT

Description ?

Create

We click on "Create" to create the credentials.

Now let us modify the pipeline script as shown below –

Script ?

```
1 pipeline {
2
3     agent any
4
5     environment {
6         NAME = 'John'
7         LASTNAME = 'Doe'
8         secretValue = credentials('SECRET_TEXT')
9     }
10
11     stages {
12         stage('Build') {
13             steps {
14                 bat 'echo "My name is %NAME% %LASTNAME%"'
15                 bat 'echo "My secret is %secretValue%"'
16                 bat 'echo "Building"'
17             }
18         }
19         stage('Test') {
20             steps {
21                 bat 'echo "Testing"'
22                 bat ''
23                 ... echo "Testing more data"
24             }
25         }
26         stage('Deploy') {
27             steps {
28                 retry(3) {
29                     bat 'echo "Deploying"'
30                 }
31                 timeout(time: 3, unit: 'SECONDS') {
32                     bat 'ping -n 1 127.0.0.1'
33                 }
34             }
35         }
36     }
37 }
38
39
```

We click on Save and hit Build Now. Then we check the console, and we will see the "My secret is *****" line, since it is a secret it will hide it while printing the value in the logs.

4.7. Post Actions

It is a part of Jenkinsfile which will run at the end of the pipeline depending on the output of the build. Post Action command is always written after the stages section in the pipeline.

A sample Post Action command is as follows –

```
post {
    always {
```



```

    echo "I will always get executed"
  }
  success {
    echo "I will be executed if the build is success"
  }
  failure {
    echo "I will be executed if the build fails"
  }
  unstable {
    echo "I will be executed if the build is unstable"
  }
}

```

A complete pipeline code will look like the below –

```

1 pipeline {
2
3   agent any
4
5   environment {
6     NAME = 'John'
7     LASTNAME = 'Doe'
8     secretValue = credentials('SECRET_TEXT')
9   }
10
11   stages {
12     stage('Build') {
13       steps {
14         bat 'echo "My name is %NAME% %LASTNAME%"'
15         bat 'echo "My secret is %secretValue%"'
16         bat 'echo "Building"'
17       }
18     }
19     stage('Test') {
20       steps {
21         bat 'echo "Testing"'
22         bat ''
23         ... echo "Testing more data"
24       }
25     }
26     stage('Deploy') {
27       steps {
28         retry(3) {
29           bat 'echo "Deploying"'
30         }
31         timeout(time: 3, unit: 'SECONDS') {
32           bat 'ping -n 1 127.0.0.1'
33         }
34       }
35     }
36   }
37
38   post {
39     always {
40       echo "I will always get executed"
41     }
42     success {
43       echo "I will be executed if the build is success"
44     }
45     failure {
46       echo "I will be executed if the build fails"
47     }
48     unstable {
49       echo "I will be executed if the build is unstable"
50     }
51   }
52 }
53

```

Once the script is modified, we click Save and hit Build Now, then we check the output. We

will see "always" and the "success" section will be executed.

```
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
I will always get executed
[Pipeline] echo
I will be executed if the build is success
[Pipeline] }
```

5. Jenkins Pipeline Example for Spring Boot Application

In this section, we are going to build a Jenkinsfile for the Spring Boot project and publish the build artifact to the Nexus OSS repository.

5.1. Pipeline Steps

Our Jenkins pipeline would consist of the following steps –

- Checkout
- Build
- Test
- Deploy


5.2. Job Setup


Go to Dashboard → New Item. Let's give our Jenkins job a name, and select Pipeline as the type.


Enter an item name


spring-jenkins


» Required field


 **Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any k


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

 **Pipeline**
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipe

 **Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multi

 **Folder**
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, as they are in different folders.

 **Multibranch Pipeline**
Creates a set of Pipeline projects according to detected branches in one SCM repository.

 **Organization Folder**
Creates a set of multibranch project subfolders by scanning for repositories.

OK

If you want to create a new item from other existing, you can use this option:

Now let's go to the Configuration section of our project.

Under General, → select GitHub project and specify the URL of our git repository.

Configure



General



Advanced Project Options



Pipeline

General

Description

[Plain text] [Preview](#)

☐ Discard old builds ?

☐ Do not allow concurrent builds

☐ Do not allow the pipeline to resume if the controller restarts

☒ GitHub project

Project url ?

<https://github.com/Bayvao/spring-jenkins-demo.git/>

Now, under the pipeline section, we will select "Pipeline script from SCM", under SCM we will select git, and specify the git repository URL from where it should fetch the Jenkinsfile because there are scenarios in real-world projects where we keep the Jenkinsfile in a separate repository.

If the Jenkinsfile repository and project repository are the same, then we mention the same URL.

Dashboard > spring-jenkins-demo > Configuration

Configure

General

Advanced Project Options

Pipeline

Pipeline

Definition

Pipeline script from SCM

SCM ?

Git

Repositories ?

Repository URL ?

https://github.com/Bayvao/spring-jenkins-demo.git

Credentials ?

- none -

+ Add

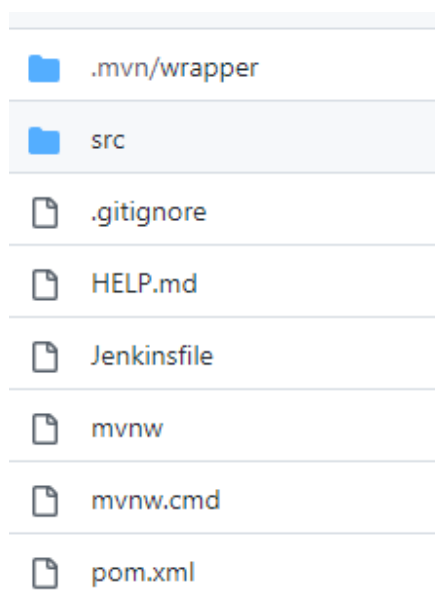
Advanced...

Add Repository

Click on Save to continue.

5.3. Creating the Jenkins File for Spring Boot App

Jenkinsfile should be present at the root path of our spring boot project.



Jenkinsfile would contain below pipeline code –



```
pipeline {
    agent any

    environment {
        mavenHome = tool 'jenkins-maven'
    }

    tools {
        jdk 'java-17'
    }

    stages {

        stage('Build'){
            steps {
                bat "mvn clean install -DskipTests"
            }
        }

        stage('Test'){
            steps{
                bat "mvn test"
            }
        }

        stage('Deploy') {
            steps {
                bat "mvn jar:jar deploy:deploy"
            }
        }
    }
}
```

5.4. Deploying the Jar File to the Nexus Maven Repository

Lets create a Maven 2 hosted repository in Nexus OSS where we will upload our artifacts. Go to the Sonatype download page (<https://help.sonatype.com/repomanager3/product-information/download>) to download, install and run Nexus on your system.

We are creating a nexus repository named "nexus-release" where we will deploy the generated artifacts. Use the below image to create a repository –

The screenshot shows the Sonatype Nexus Repository Manager interface. The left sidebar contains the 'Administration' menu with options like Repository, Blob Stores, Cleanup Policies, Content Selectors, Proprietary Repositories, Routing Rules, Security, Privileges, Roles, Users, Anonymous Access, LDAP, Realms, SSL Certificates, IQ Server, Support, Logging, Log Viewer, Status, Support ZIP, System Information, and System. The main panel is titled 'Repositories' and shows the configuration for a new repository named 'nexus-release'. The configuration includes fields for Name, Online status, Version policy (Release), Layout policy (Strict), Content Disposition (Inline), Blob store (default), Strict Content Type Validation (checked), Deployment policy (Disable redeploy), Proprietary Components (unchecked), and Cleanup Policies (Available and Applied lists).

In the Deployment Policy we have used the “Disable redeploy” option which basically means that once a version of the artifact has been published/deployed in Nexus, the same version won’t be deployed again, nexus won’t allow it. Once the above-shown configuration is done, click “Create Repository”.

The next configuration is configuring the settings.xml file present in the maven folder to point to the nexus server inside your local machine “maven/conf/settings.xml” file. Inside the <server></server> tag we will specify our nexus username, password and id for use in your application.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.2.0 http://maven.a
```

```
<pluginGroups>
```

```

</pluginGroups>

<proxies>
</proxies>

<servers>
  <server>
    <id>nexus-release</id>
    <username>admin</username>
    <password>admin</password>
  </server>
</servers>

<mirrors>
  <mirror>
    <id>maven-default-http-blocker</id>
    <mirrorOf>external:http:*</mirrorOf>
    <name>Pseudo repository to mirror external repositories initially using HTTP.<
    <url>http://0.0.0.0/</url>
    <blocked>true</blocked>
  </mirror>
</mirrors>

<profiles>
</profiles>
  <activeProfiles>
</activeProfiles>
</settings>

```

Once the above configuration is done, next is configuring the repository manager in the project *pom.xml* file.

```

<distributionManagement>
  <repository>
    <uniqueVersion>>false</uniqueVersion>
    <id>nexus-release</id>
    <name>nexus-release</name>
    <url>http://localhost:8081/repository/nexus-release/</url>
  </repository>
</distributionManagement>

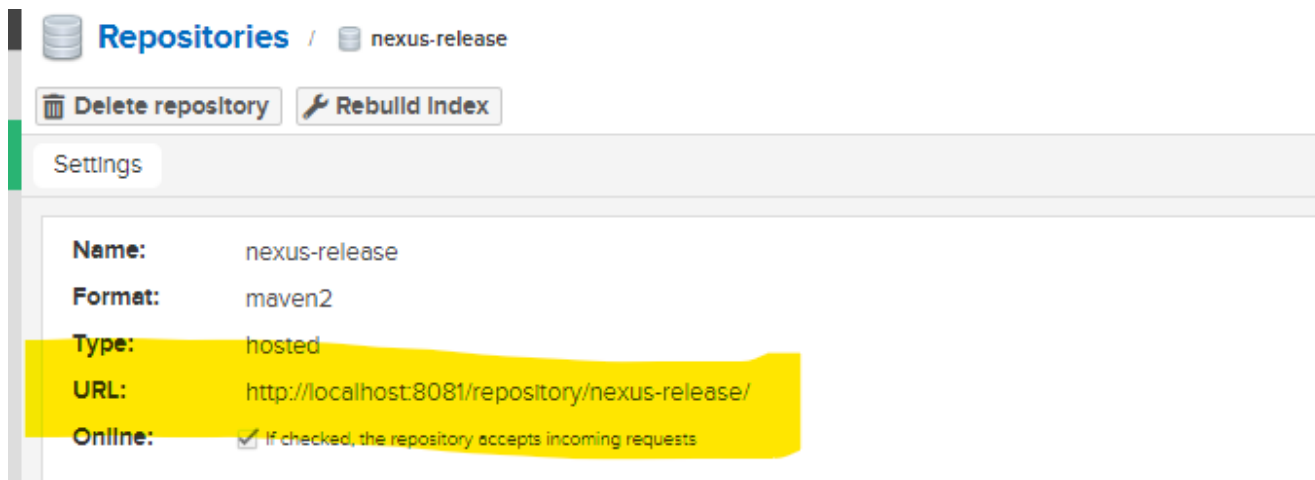
```

Here inside **<distributionManagement>** tag we specify the **id** which we used in the settings.xml **<server>** tag, and then we specify the repository URL to where it should deploy.

Using the id of the server, it will fetch the username and password for nexus during the

deployment phase of the pipeline.

The repository URL will be found here –



Once all the final configuration is done.

When we hit Build Now, and check the console output, we will see the jar file has been built, test cases are executed and finally if the build is successful, we will see the generated jar is uploaded to the nexus repository.

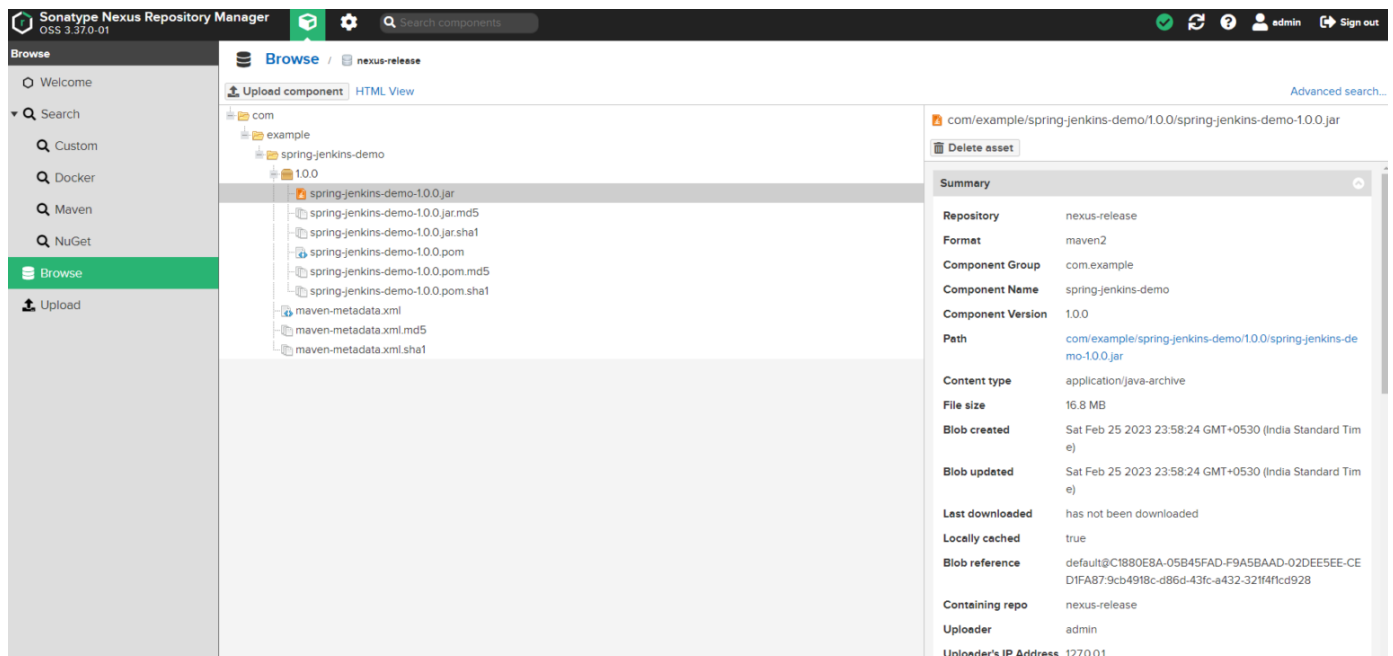
```
Uploaded to nexus-release: http://localhost:8081/repository/nexus-release/com/example/spring-jenkins-demo/1.0.0/spring-jenkins-demo-1.0.0.jar (18 MB at 8.9 MB/s)
Uploading to nexus-release: http://localhost:8081/repository/nexus-release/com/example/spring-jenkins-demo/1.0.0/spring-jenkins-demo-1.0.0.pom
Progress (1): 3.8 kB

Uploaded to nexus-release: http://localhost:8081/repository/nexus-release/com/example/spring-jenkins-demo/1.0.0/spring-jenkins-demo-1.0.0.pom (3.8 kB at 8.1 kB/s)
Downloading from nexus-release: http://localhost:8081/repository/nexus-release/com/example/spring-jenkins-demo/maven-metadata.xml
Uploading to nexus-release: http://localhost:8081/repository/nexus-release/com/example/spring-jenkins-demo/maven-metadata.xml
Progress (1): 310 B

Uploaded to nexus-release: http://localhost:8081/repository/nexus-release/com/example/spring-jenkins-demo/maven-metadata.xml (310 B at 301 B/s)
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.901 s
[INFO] Finished at: 2023-02-25T23:58:27+05:30
[INFO] -----
```

5.4. Verify the Deployed Jar

Finally, let's go to the Nexus OSS, and let's check the uploaded jar file.



We see that the jar file and the pom.xml file has been successfully uploaded in the Nexus Repository Manager.

6. Conclusion

In this Jenkins and Spring boot tutorial, we learned:

- how to build jar/artifacts manually through Jenkins UI.
- Learned about Pipeline and how to write a pipeline in Jenkins.
- Learned about Nexus OSS repository and how to deploy build artifacts in nexus.
- Finally, we learned about Jenkinsfile and we wrote an end-to-end Jenkinsfile for a Spring boot project.

Happy Learning !!

Weekly Newsletter

Stay Up-to-Date with Our Weekly Updates. Right into Your Inbox.

Email Address