## Jenkins Tutorial

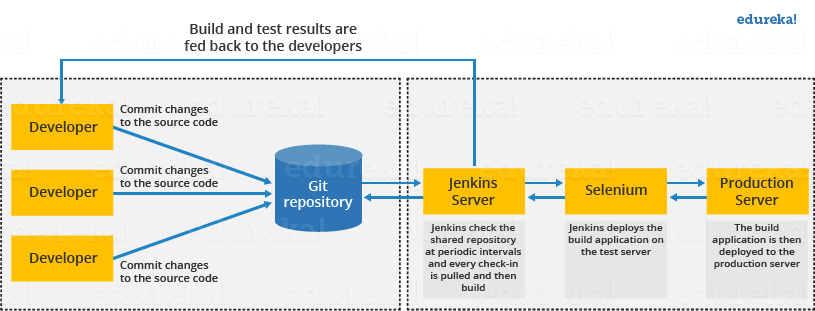
Jenkins is one of the most important tools in [DevOps](https://www.edureka.co/devops" \t "_blank). I hope you have read my previous blog on [What is Jenkins](https://www.edureka.co/blog/what-is-jenkins/). In this Jenkins Tutorial blog, I will focus on Jenkins architecture and Jenkins build pipeline along with that I will show you how to create a build in Jenkins.

* Jenkins is used to integrate all DevOps stages with the help of plugins.
* Commonly used Jenkins plugins are Git, Amazon EC2, Maven 2 project, HTML publisher etc.
* Jenkins has well over 1000 plugins and 147,000 active installations along with over 1 million users around the world.
* With Continuous Integration every change made in the source code is built. It performs other functions as well, that depends on the tool used for Continuous Integration.
* Nokia shifted from Nightly build to Continuous Integration.
* Process before Continuous Integration had many flaws. As a result, not only the software delivery was slow but the quality of software was also not up to the mark. Developers also had a tough time in locating and fixing bugs.
* Continuous Integration with Jenkins overcame these shortcomings by continuously triggering a build and test for every change made in the source code.

Now is the correct time to understand Jenkins architecture.

## Jenkins Architecture

.



This single Jenkins server was not enough to meet certain requirements like:

* Sometimes you might need several different environments to test your builds. This cannot be done by a single Jenkins server.
* If larger and heavier projects get built on a regular basis then a single Jenkins server cannot simply handle the entire load.

To address the above stated needs, Jenkins distributed architecture was introduced.

## Jenkins Distributed Architecture

Jenkins uses a Master-Slave architecture to manage distributed builds. In this architecture, Master and Slave communicate through TCP/IP protocol.

**Jenkins Master**

Your main Jenkins server is the Master. The Master’s job is to handle:

Scheduling build jobs.

Dispatching builds to the slaves for the actual execution.

Monitor the slaves (possibly taking them online and offline as required).

Recording and presenting the build results.

A Master instance of Jenkins can also execute build jobs directly.

Jenkins Slave

A Slave is a Java executable that runs on a remote machine. Following are the characteristics of Jenkins Slaves:

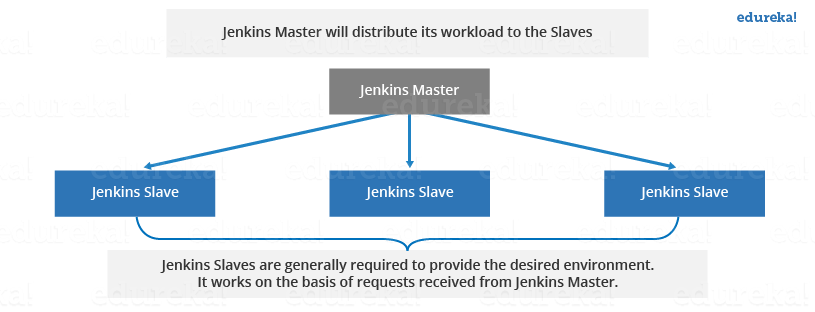
It hears requests from the Jenkins Master instance.

Slaves can run on a variety of operating systems.

The job of a Slave is to do as they are told to, which involves executing build jobs dispatched by the Master.

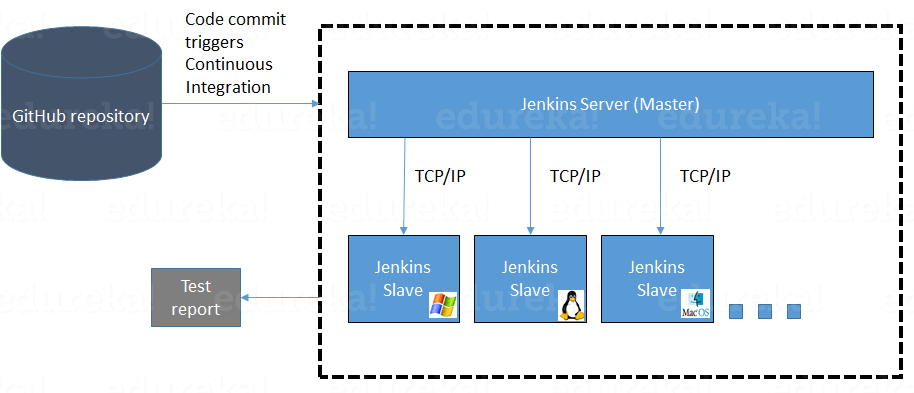
You can configure a project to always run on a particular Slave machine, or a particular type of Slave machine, or simply let Jenkins pick the next available Slave.

The diagram below is self explanatory. It consists of a Jenkins Master which is managing three Jenkins Slave.



Now let us look at an example in which Jenkins is used for testing in different environments like: Ubuntu, MAC, Windows etc.

The diagram below represents the same:

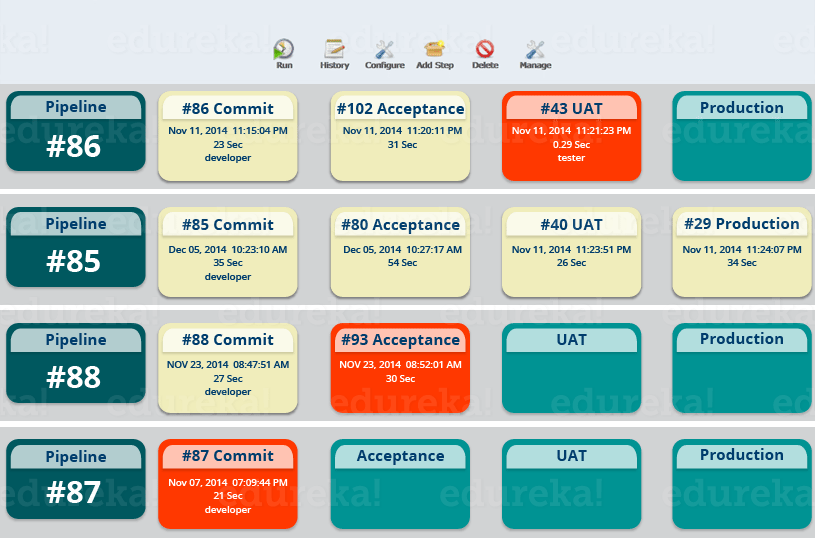


The following functions are performed in the above image:

* Jenkins checks the Git repository at periodic intervals for any changes made in the source code.
* Each builds requires a different testing environment which is not possible for a single Jenkins server. In order to perform testing in different environments Jenkins uses various Slaves as shown in the diagram.
* Jenkins Master requests these Slaves to perform testing and to generate test reports.

## Jenkins Build Pipeline

It is used to know which task Jenkins is currently executing. Often several different changes are made by several developers at once, so it is useful to know which change is getting tested or which change is sitting in the queue or which build is broken. This is where pipeline comes into picture. The Jenkins Pipeline gives you an overview of where tests are up to. In build pipeline the build as a whole is broken down into sections, such as the unit test, acceptance test, packaging, reporting and deployment phases. The pipeline phases can be executed in series or parallel, and if one phase is successful, it automatically moves on to the next phase (hence the relevance of the name “pipeline”).The below image shows how a multiple build Pipeline looks like.



Hope you have understood the theoretical concepts. Now, let’s have some fun with hands-on.

I will create a new job in Jenkins, it is a **Freestyle Project**. However, there are 3 more options available. Let us look at the types of build jobs available in Jenkins.

**Freestyle Project:**

Freestyle build jobs are general-purpose build jobs, which provides maximum flexibility. The freestyle build job is the most flexible and configurable option, and can be used for any type of project. It is relatively straightforward to set up, and many of the options we configure here also appear in other build jobs.

 Multiconfiguration Job:

The “multiconfiguration project” (also referred to as a “matrix project”) allows you run the same build job on different environments. It is used for testing an application in different environments, with different databases, or even on different build machines.

Monitor an External Job:

The “Monitor an external job” build job lets you keep an eye on non-interactive processes, such as cron jobs.

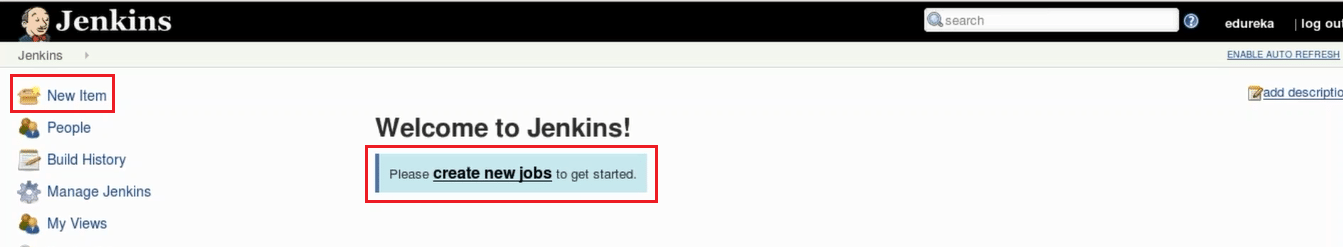
Maven Project:

The “maven2/3 project” is a build job specially adapted to Maven projects. Jenkins understands Maven pom files and project structures, and can use the information gleaned from the pom file to reduce the work you need to do to set up your project.

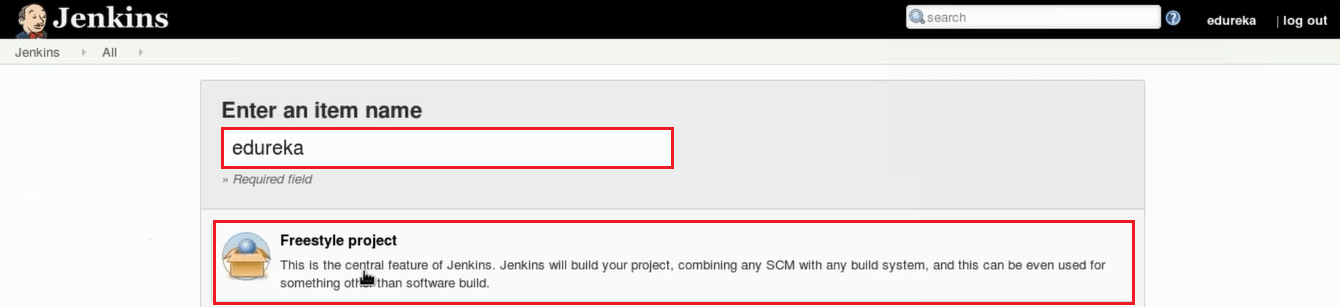
Here is a video on Jenkins tutorial for better understanding of Jenkins. Check out this Jenkins tutorial video.

## Creating a Build Using Jenkins

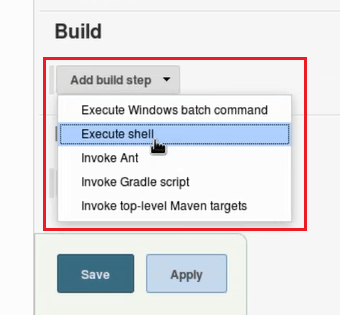
**Step 1:** From the Jenkins interface home, select **New Item.**

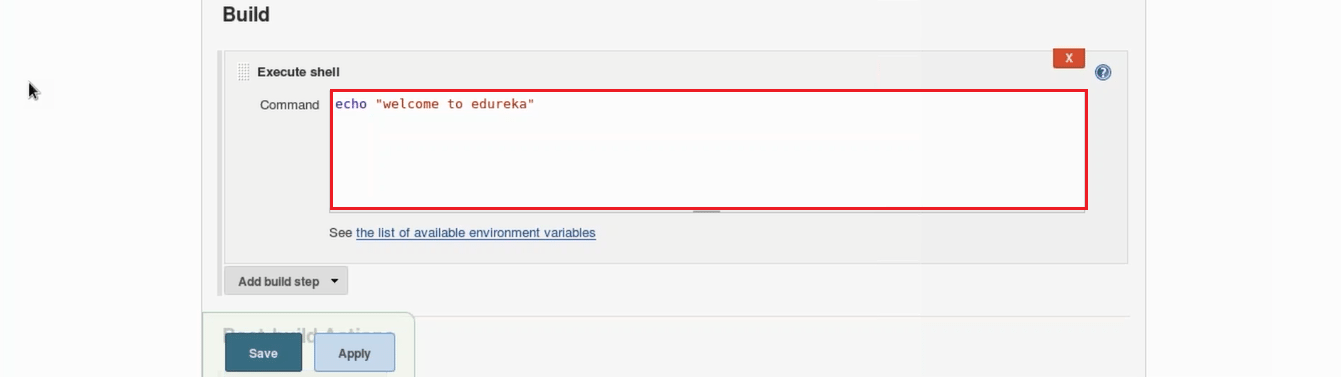


Step 2: Enter a name and select Freestyle project.

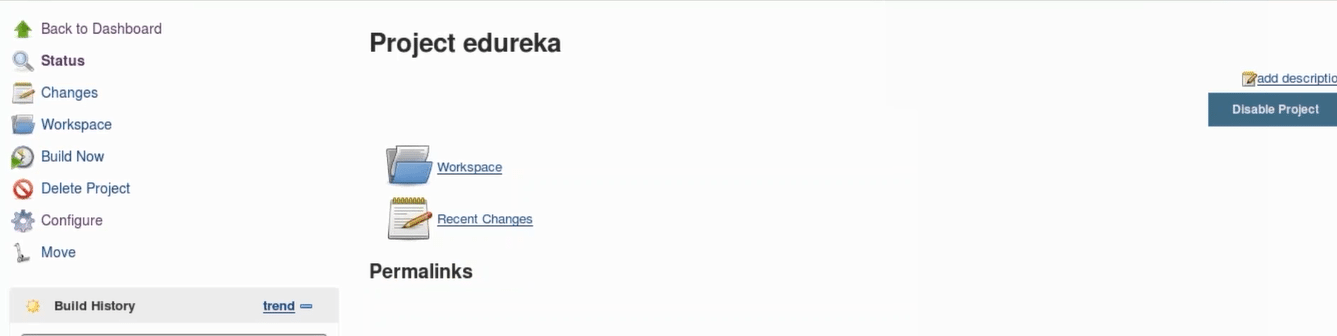


Step 3: This next page is where you specify the job configuration. As you’ll quickly observe, there are a number of settings available when you create a new project. On this configuration page, you also have the option to Add build step to perform extra actions like running scripts. I will execute a shell script.

This will provide you with a text box in which you can add whatever commands you need. You can use scripts to run various tasks like server maintenance, version control, reading system settings, etc. I will use this section to run a simple script.



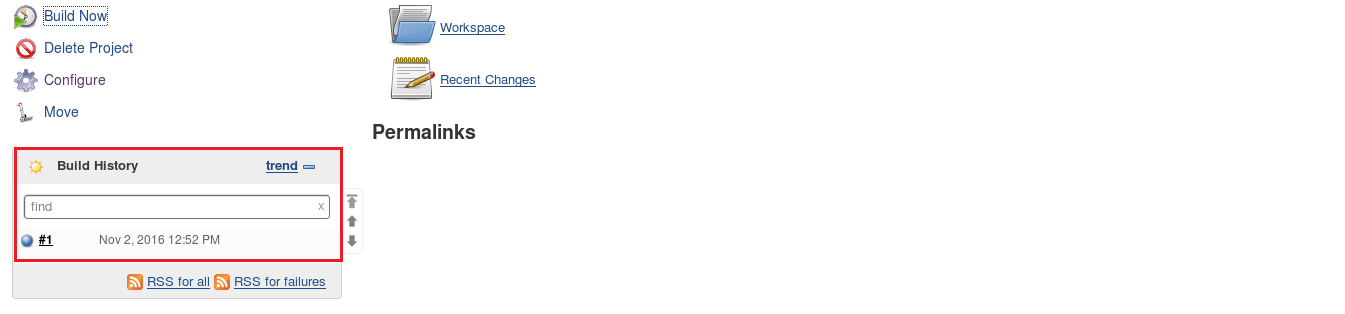
Step 4: Save the project, and you’ll be taken to a project overview page. Here you can see information about the project, including its built history.



Step 5: Click Build Now on the left-hand side to start the build.



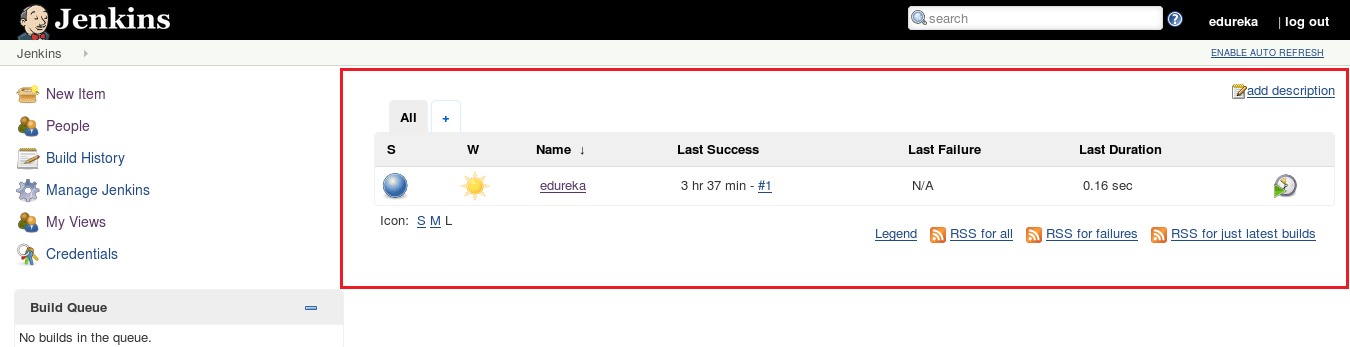
Step 6: To see more information, click on that build in the build history area, whereupon you’ll be taken to a page with an overview of the build information.



Step 7: The Console Output link on this page is especially useful for examining the results of the job in detail.



Step 8: If you go back to Jenkins home, you’ll see an overview of all projects and their information, including status.



Status of the build is indicated in two ways, by a weather icon and by a colored ball. The weather icon is particularly helpful as it shows you a record of multiple builds in one image.

As you can see in the above image, the sun represents that all of my builds were successful. The color of the ball gives us the status of that particular build, in the above image the color of the ball is blue which means that this particular build was successful.

In this Jenkins Tutorial, I have just given an introductory example. In my next blog, I will show you how to pull and build code from the GitHub repository using Jenkins.

What is Continuous Integration? -Continuous Integration using Jenkins

Continuous Integration is a development practice where developers integrate code into a shared repository frequently where each integration is verified by an automated build and automated tests. It is the most important part of *[DevOps](https://www.edureka.co/blog/devops-tutorial?utm_source=medium&utm_medium=content-link&utm_campaign=continuous-integration" \t "_blank)* that is used to integrate various DevOps stages. In this blog, we will deal with the problems developers face while writing, testing and delivering software to end users and how they solve it using CI.

In this blog, we will focus on the below topics:

1. Traditional Integration
2. Problems with Traditional Integration
3. What is Continuous Integration?
4. Benefits of Continuous Integration
5. Requirements for CI System
6. What is Jenkins — The Ultimate CI Tool
7. Demo on Continuous Integration Using Jenkins

**Traditional Integration**

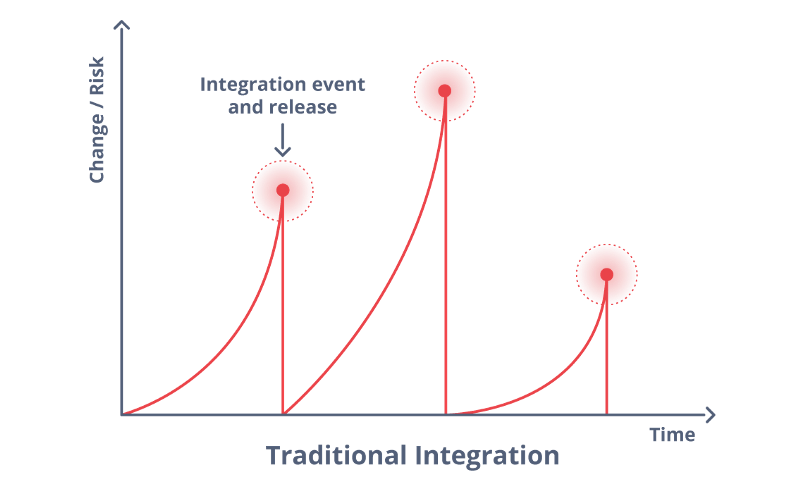
In Traditional Integration or/software development cycle,

* Each developer gets a copy of the code from the central repository.
* All developers begin at the same starting point and work on it.
* Each developer makes progress by working on their own or in a team.
* They add or change classes, methods, and functions, shaping the code to meet their needs, and eventually, they complete the task they were assigned to do.
* Meanwhile, the other developers and teams continue working on their own tasks, changing the code or adding new code, solving the problems they have been assigned.

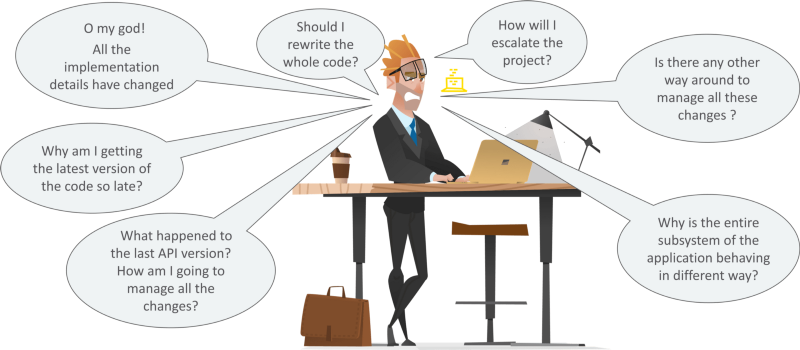
If we take a step back and look at the big picture, i.e. the entire project, we can see that all developers working on a project are changing the context for the other developers as they are working on the source code.

The main factors that can make these problems escalate:

* The size of the team working on the project.
* The amount of time passed since the developer got the latest version of the code from the central repository.

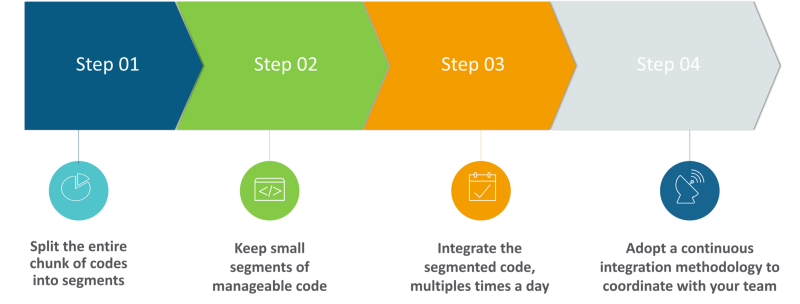


**Problems With Traditional Integration**



### What’s the Solution for Problems faced in Traditional Integration?

So, below are the steps to solve the above problems:

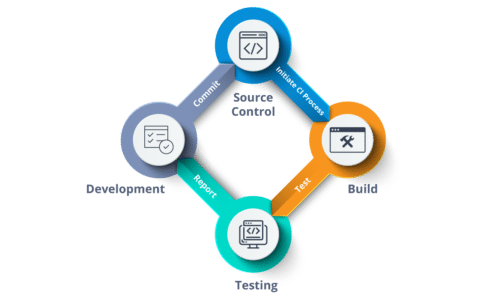


So, let us see what exactly is Continuous Integration?

### What is Continuous Integration?

**We will Start with the Martin Fowler’s Definition:**

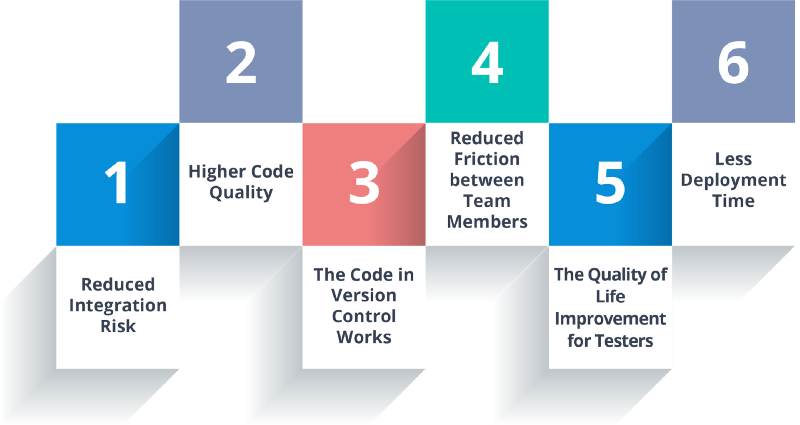
Continuous Integration is a software development practice where members of a team integrate their work frequently, usually, each person integrates at least daily leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly*.*



Automating your build, test and deploy processes can increase the problems commonly happening on projects. So we should have a reliable method of integrating that will ensure that the errors can be found sooner than later.

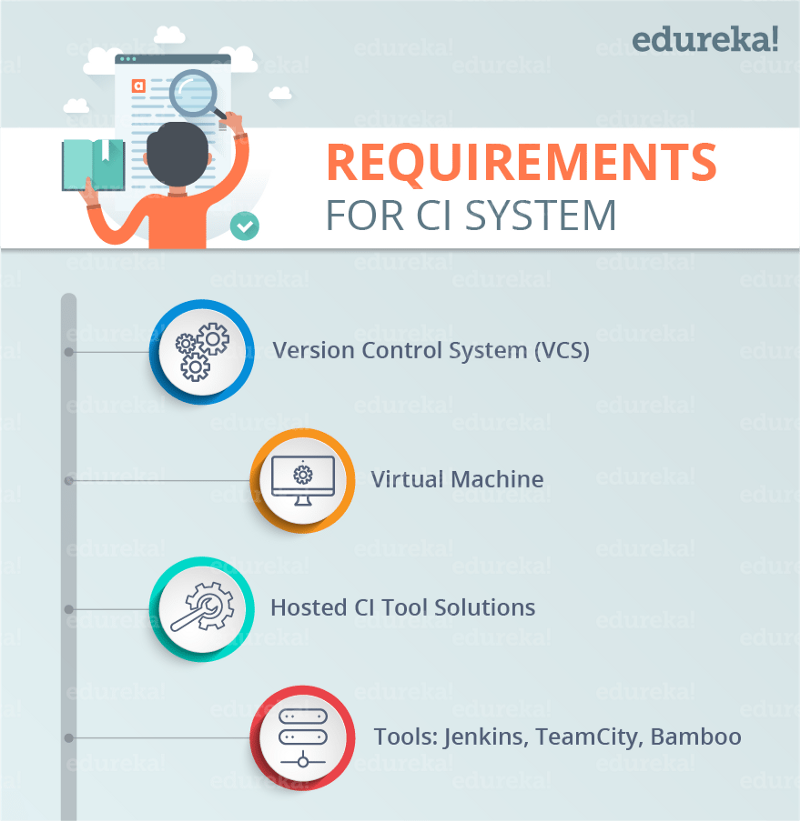
Let’s move on and see what are the benefits of Continuous Integration.

### Benefits of Continuous Integration



1. **Reduced Integration Risk:** Often, working on projects means multiple people are working on the separate tasks or parts of the code which makes it risky to integrate. Debugging and solving the issue can be really painful and can potentially mean a lot of changes to the code. Integrating more frequently can help reduce these kinds of problems to a minimum.
2. **Higher Code Quality:** Focusing more on the functionality of the code results in a higher quality product.
3. **The Code in Version Control works:** If you commit something that breaks the build, you and your team get the notice immediately and the problem is fixed before anyone else pulls the “broken” code.
4. **Reduced friction between team members:** Having the impartial system in place reduces the frequency of quarrels among team members.
5. **Easy for QA Team:** Having different versions and builds of the code can help isolate and trace bugs efficiently, and it makes life easier for the QA team.
6. **Less time deploying:** Deploying projects can be very tedious and time-consuming, and automating that process makes perfect sense.

### Requirements for CI System



But you might be wondering what are the requirements for the installing of the CI system for your needs. If you want to install CI server in your own environment, you’ll need a few things first.

* **Version control system (VCS)**. It provides a reliable method to centralize and preserve changes made to your project over time.
* **Virtual Machine:** For onsite solutions, you should have a spare server or at least a virtual machine. for a clean machine to build your system on is of the essential importance.
* **Hosted CI Tool solutions:** To avoid servers or virtual machines, you can go for hosted CI tool solutions that help in the maintenance of the whole process and offers easier scalability.
* **Tools:**If you opt-in for the self-hosted variant, you will need to install one of the many available continuous integration tools like Jenkins, TeamCity, Bamboo etc.

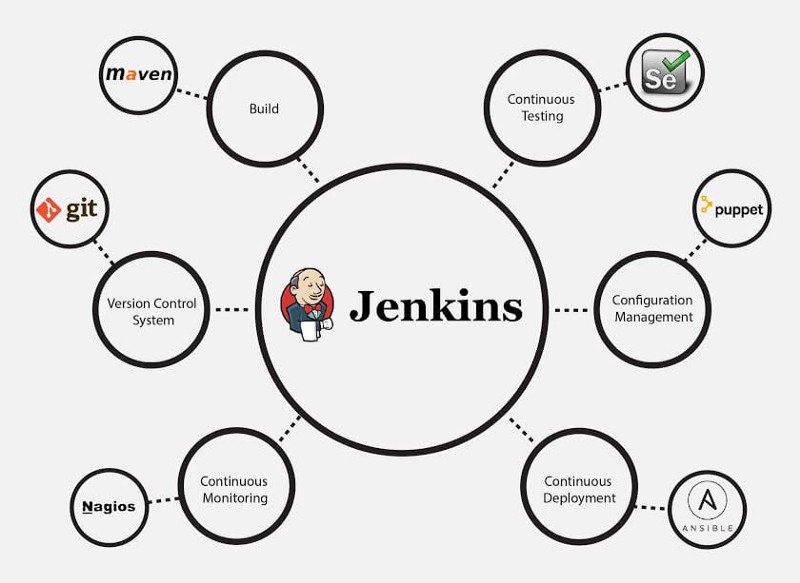
**What is Jenkins — The Ultimate CI Tool**

[Jenkins](https://www.edureka.co/blog/what-is-jenkins/?utm_source=medium&utm_medium=content-link&utm_campaign=continuous-integration) is an open source automation tool written in Java with plugins built for Continuous Integration purpose. Jenkins is used to build and test your software projects continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. It also allows you to continuously deliver your software by integrating with a large number of testing and deployment technologies.

With Jenkins, organizations can accelerate the software development process through automation. Jenkins integrates development life-cycle processes of all kinds, including build, document, test, package, stage, deploy, static analysis and much more.

Jenkins achieves Continuous Integration with the help of plugins. Plugins allow the integration of Various DevOps stages. If you want to integrate a particular tool, you need to install the plugins for that tool. For example [Git](https://www.edureka.co/blog/what-is-git/?utm_source=medium&utm_medium=content-link&utm_campaign=continuous-integration" \t "_blank), Maven 2 project, [Amazon EC2](https://www.edureka.co/blog/ec2-aws-tutorial-elastic-compute-cloud/?utm_source=medium&utm_medium=content-link&utm_campaign=continuous-integration), HTML publisher etc.

The image below depicts that Jenkins is integrating various DevOps stages:

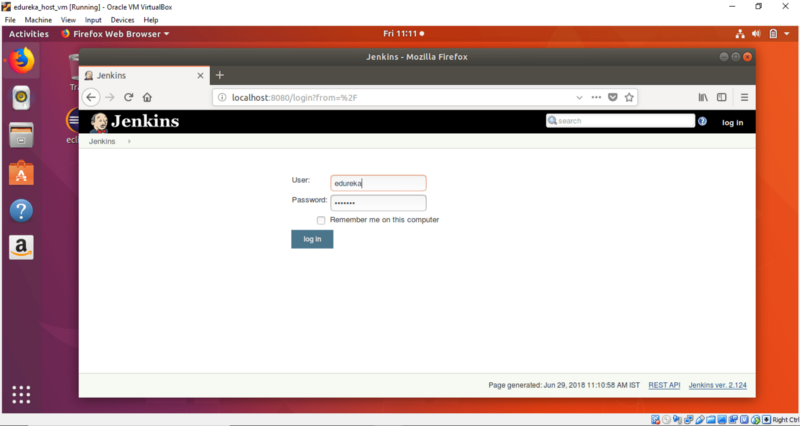


### Demo on Continuous Integration Using Jenkins

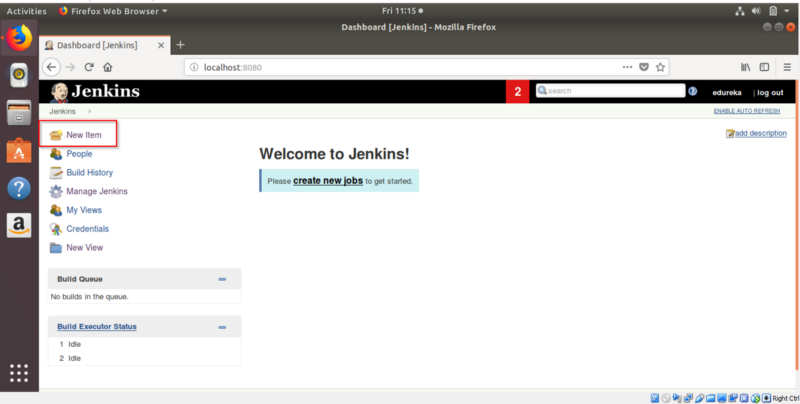
A company named Sanders & Fresco Private Ltd. got the project from a client which consisted of 4 modules. The Project manager divided these modules between two developers. Now to maintain the consistency in the flow of the project, they built a [*Continuous Delivery Pipeline*](https://www.edureka.co/blog/continuous-delivery?utm_source=medium&utm_medium=content-link&utm_campaign=continuous-integration) where they executed all the modules in terms of jobs (a single job will have two modules). After building those jobs, they synced it in a pipeline. They also checked the build at regular intervals of time while they were developing their project. So, the path of the project which was in git hub repository was provided through Jenkins. Using the concept of CI, they built their project after a fixed interval of time which can be defined by build trigger option in Jenkins.

So, let us see how we can do this using Jenkins.

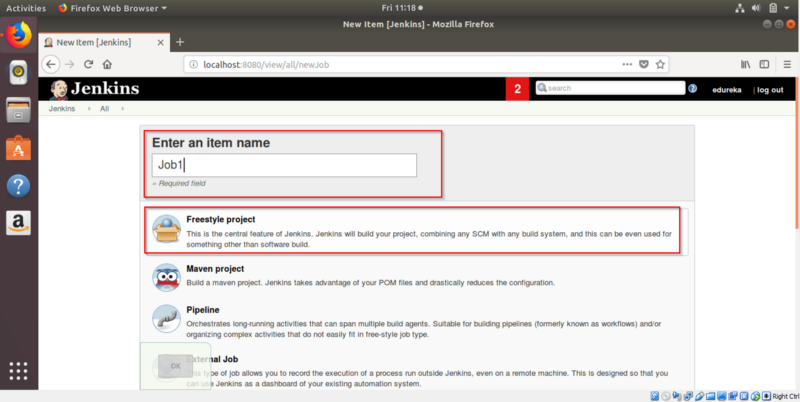
**Step 1:** Open the Jenkins on your designated port number in your VM.



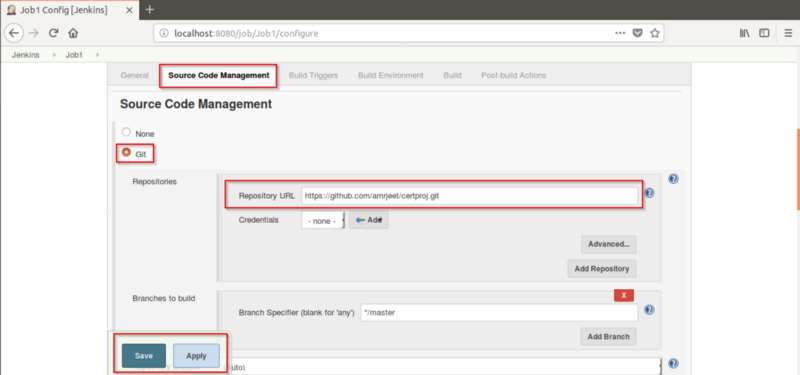
**Step 2:** Click on New Item to create a new Job.



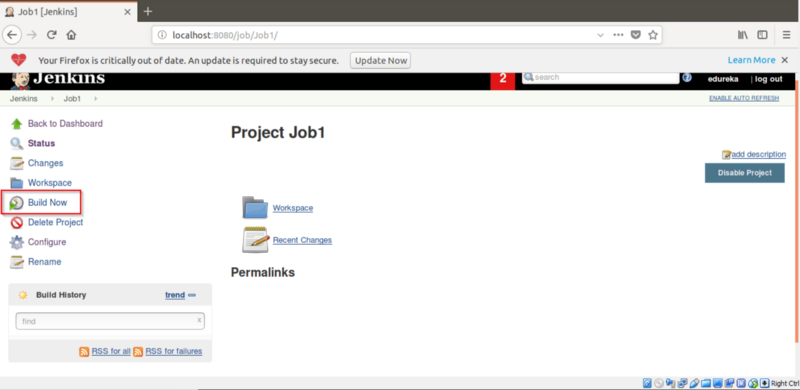
**Step 3:** Give the name of Freestyle Project. Here I have given **Job1.**



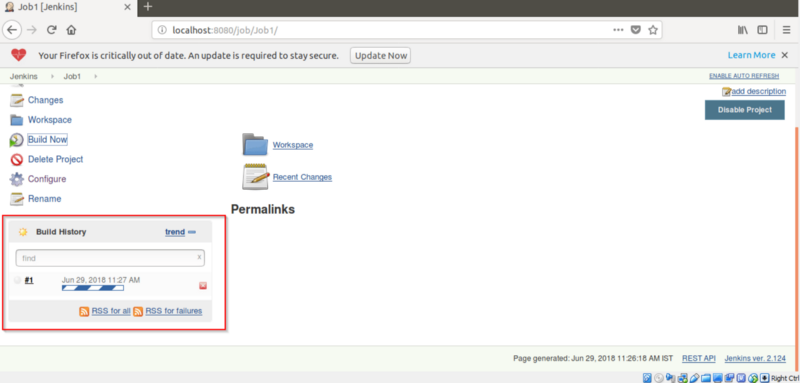
**Step 4:** Now go to **Source Code Management-> Git**. Fill the project path to pull the project from Git and Click on Apply.



Step 5:**Click on Build Now.**

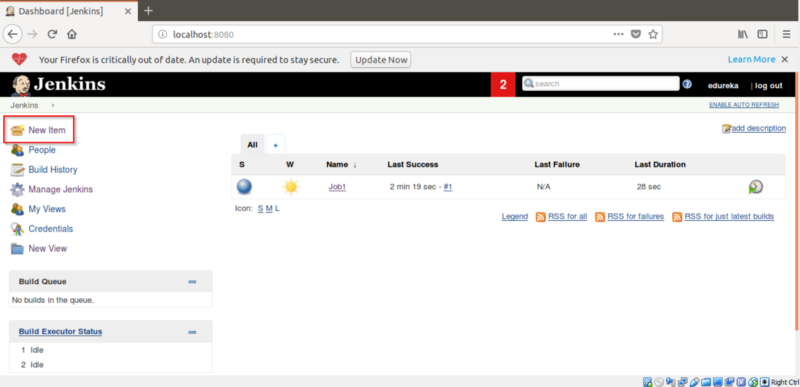


**Step 6:** The Job1 will start building it.

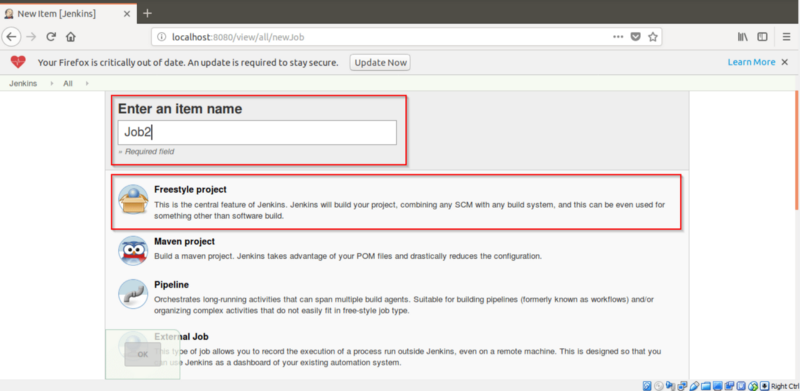


Now we will create a project/Job2 which will build after a fixed interval of time (i.e. after every minute) which we can define by using Poll SCM.

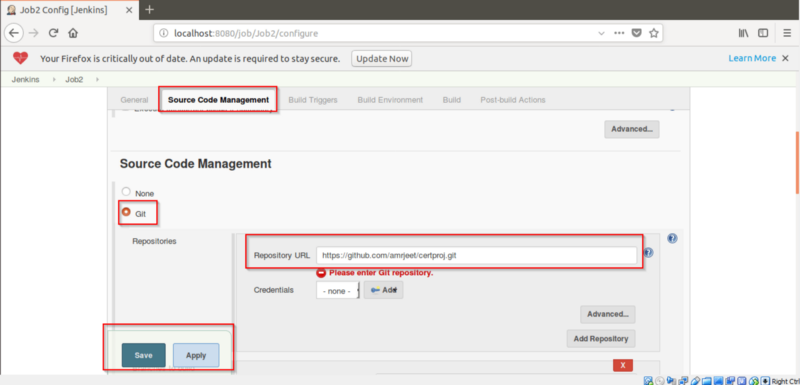
**Step 7:** Click on New Item to create a new Job.



Step 8:**Give the name of Freestyle Project. Here I have given Job2.**



**Step 9:** Now go to **Source Code Management-> Git**. Fill the project path to pull the project from Git and Click on Apply.



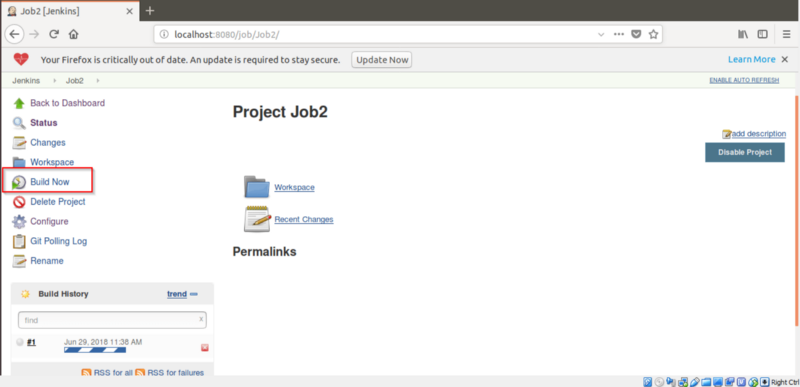
Step 10:**Then go to**Build Trigger -> Poll SCM**. Enter ” \* \* \* \* \* ” to build the Job2 after every minute and click on Apply.**



After clicking on Apply, a message will be displayed. Click on **Save** button.

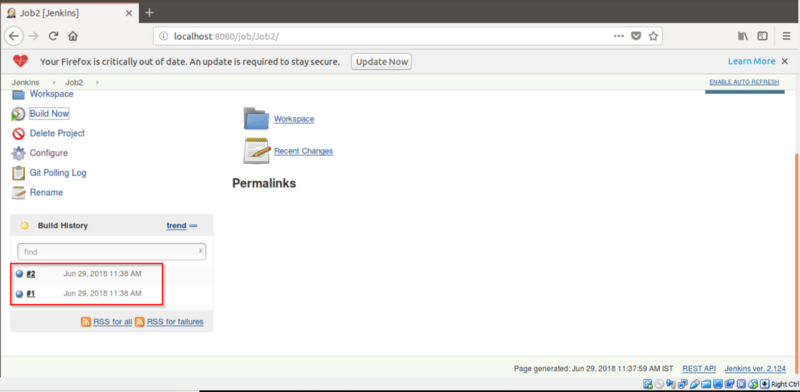


**Step 11:** Click on Build Now.



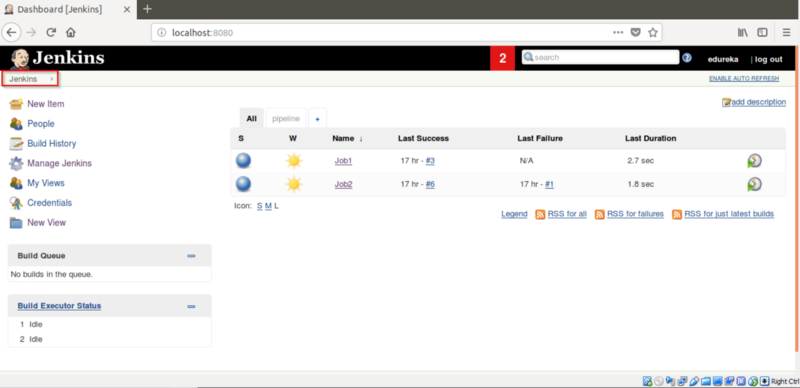
Step 12: **The Job2 will start building after every minute.**

**Note:**To check for build status, Click on **Git Polling Log.**This will show details of Job2 i.e. the time at which it got built.

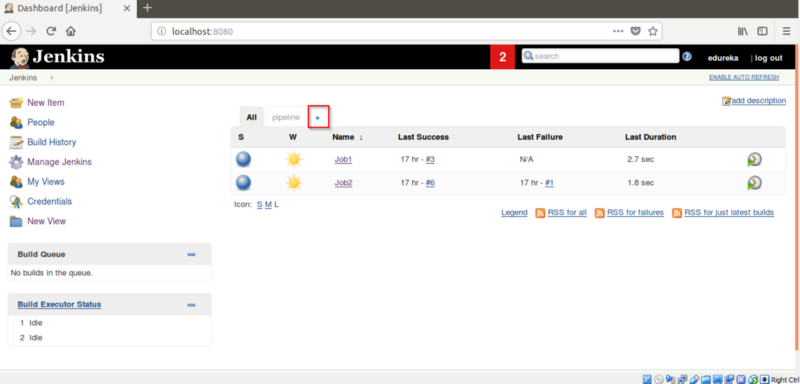


Now, we will be building a Jenkins Pipeline to show Continuous Delivery.

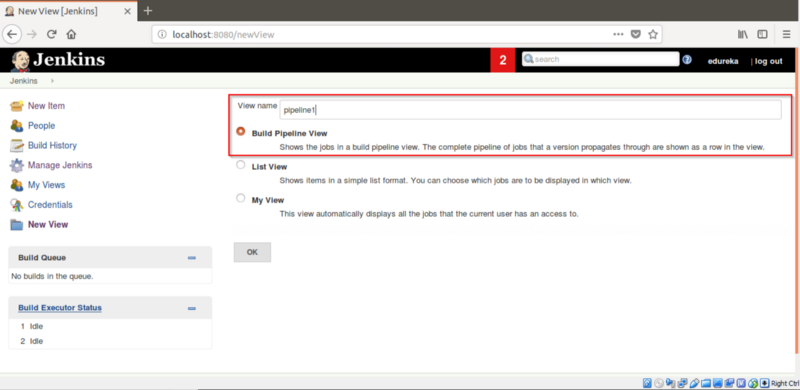
Step 13: **Now go to Jenkins homepage by clicking on Jenkins.**



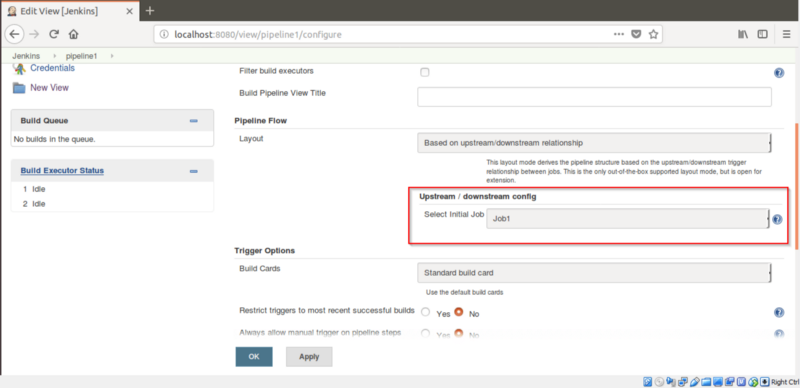
**Step 14:**Click on “+” to create a pipeline view.



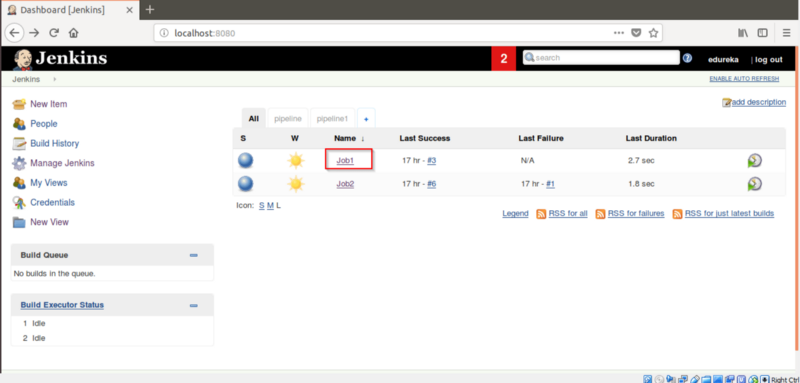
**Step 15:**Name the view and check mark **“Build Pipeline** View” and Click on OK.



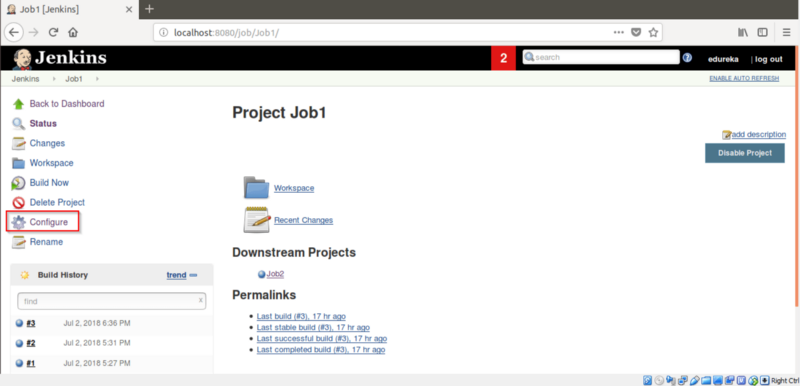
Step 16: **Go to**Pipeline Flow **and add**Job1 **in**Select Initial Job. **Click on OK.**



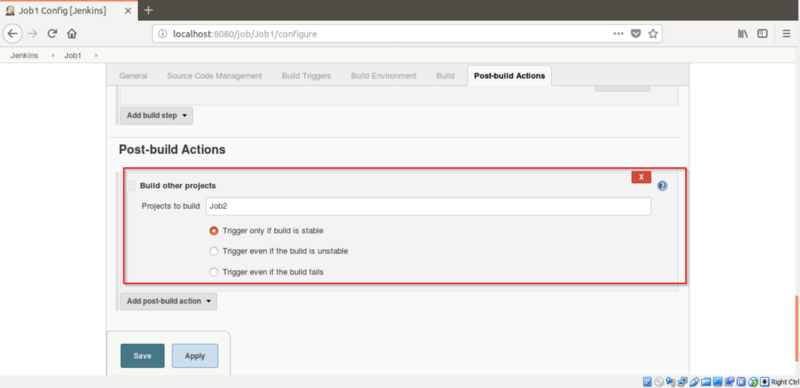
**Step 17:**Now Click on Job1.



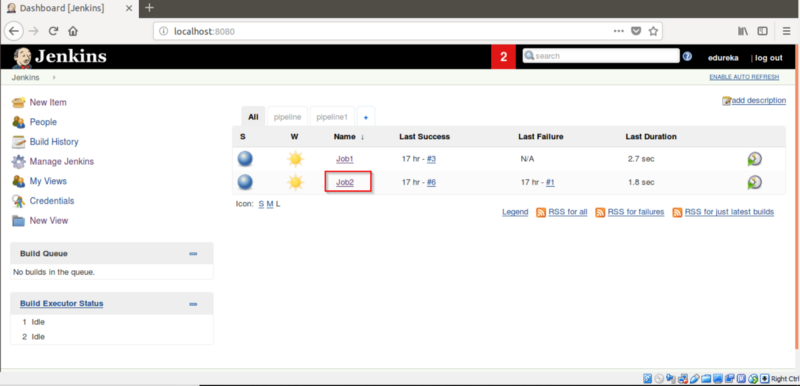
Step 18: **Then Click on**Configure**.**



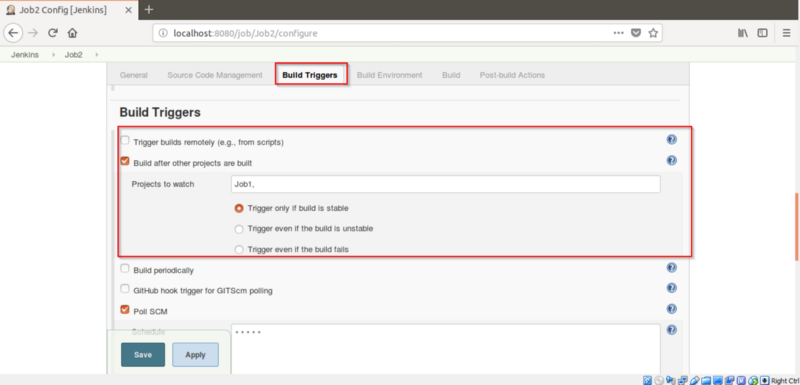
**Step 19:**Go to **Post-build Actions**and select **Build other projects.**Add Job2 in Projects to build and checkmark the “**Trigger only if the build is stable**“. Click on **Save**.



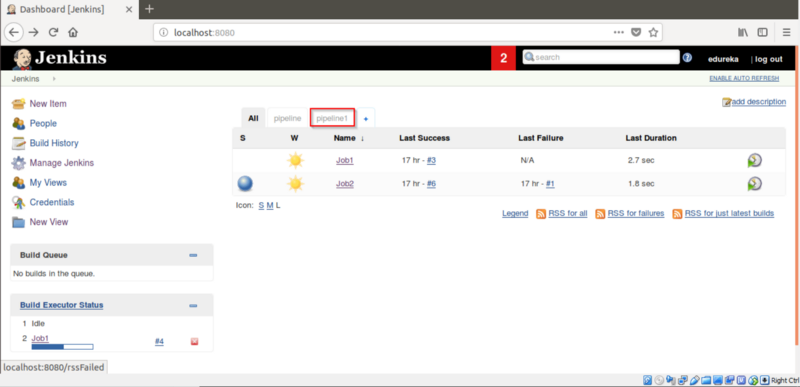
Step 20: **Now Click on Job2 and click on**Configure**.**



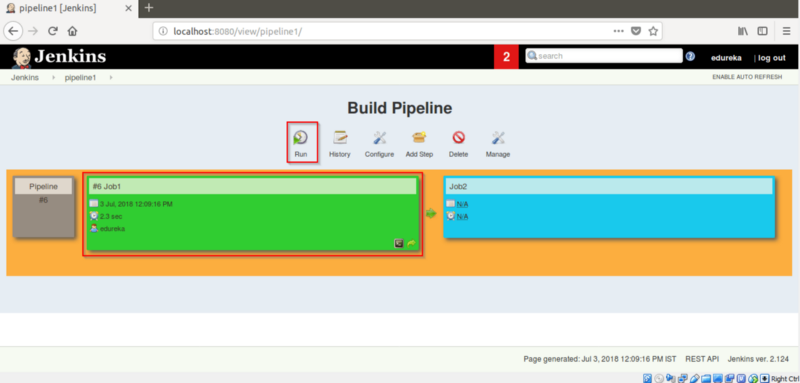
**Step 21:**Go to **Build Triggers ->**Select **Build after other projects are built.**Add Job1 in Projects to watch label.



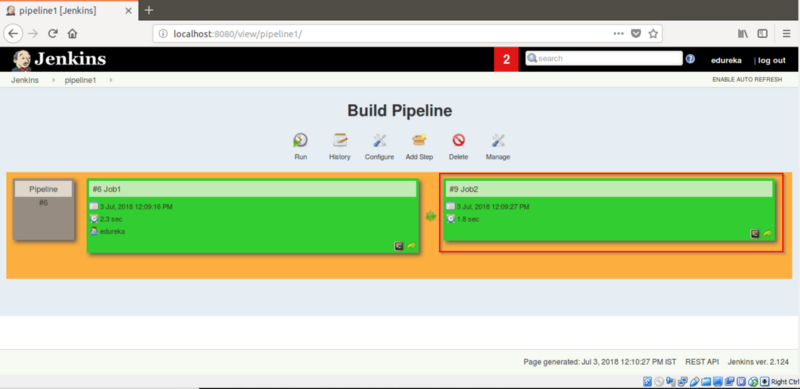
Step 22: **Click on the**pipeline1**view you created.**



**Step 23:**Click on **Run.**Job1 will start building.



**After building Job1, Job2 will automatically build itself.**



**So, this was all about the What Is Continuous Integration and its implementation using Jenkins.**

## Continuous Delivery

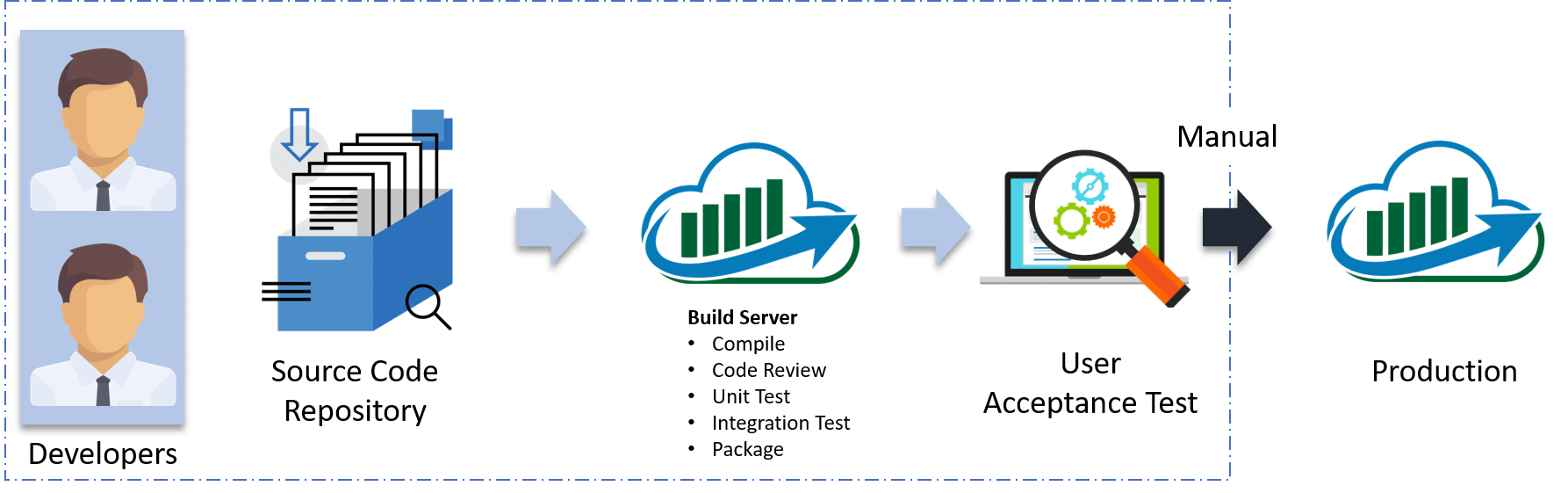
Continuous Delivery is a process, where code changes are automatically built, tested, and prepared for a release to production.  What is Continuous Delivery?

* Types of Software Testing
* Difference Between Continuous Integration, Delivery, and Deployment
* What is the need for Continuous Delivery?
* Hands-on Using Jenkins and Tomcat

Let us quickly understand how Continuous Delivery works.

## What Is Continuous Delivery?

It is a process where you build software in such a way that it can be released to production at any time. Consider the diagram below:



Let me explain the above diagram:

* Automated build scripts will detect changes in Source Code Management (SCM) like Git.
* Once the change is detected, source code would be deployed to a dedicated build server to make sure build is not failing and all test classes and integration tests are running fine.
* Then, the build application is deployed on the test servers (pre-production servers) for User Acceptance Test (UAT).
* Finally, the application is manually deployed on the production servers for release.

Before I proceed, it will only be fair I explain to you the different types of testing.

## Types of Software Testing:

Broadly speaking there are two types of testing:

* **Blackbox Testing:** It is a testing technique that ignores the internal mechanism of the system and focuses on the output generated against any input and execution of the system. It is also called functional testing. It is basically used for validating the software.
* **Whitebox Testing:** is a testing technique that takes into account the internal mechanism of a system. It is also called structural testing and glass box testing. It is basically used for verifying the software.

### Whitebox testing:

There are two types of testing, that falls under this category.

* **Unit Testing:** It is the testing of an individual unit or group of related units. It is often done by the programmer to test that the unit he/she has implemented is producing expected output against given input.
* **Integration Testing:**It is a type of testing in which a group of components are combined to produce the output. Also, the interaction between software and hardware is tested if software and hardware components have any relation. It may fall under both white box testing and black box testing.

### Blackbox Testing:

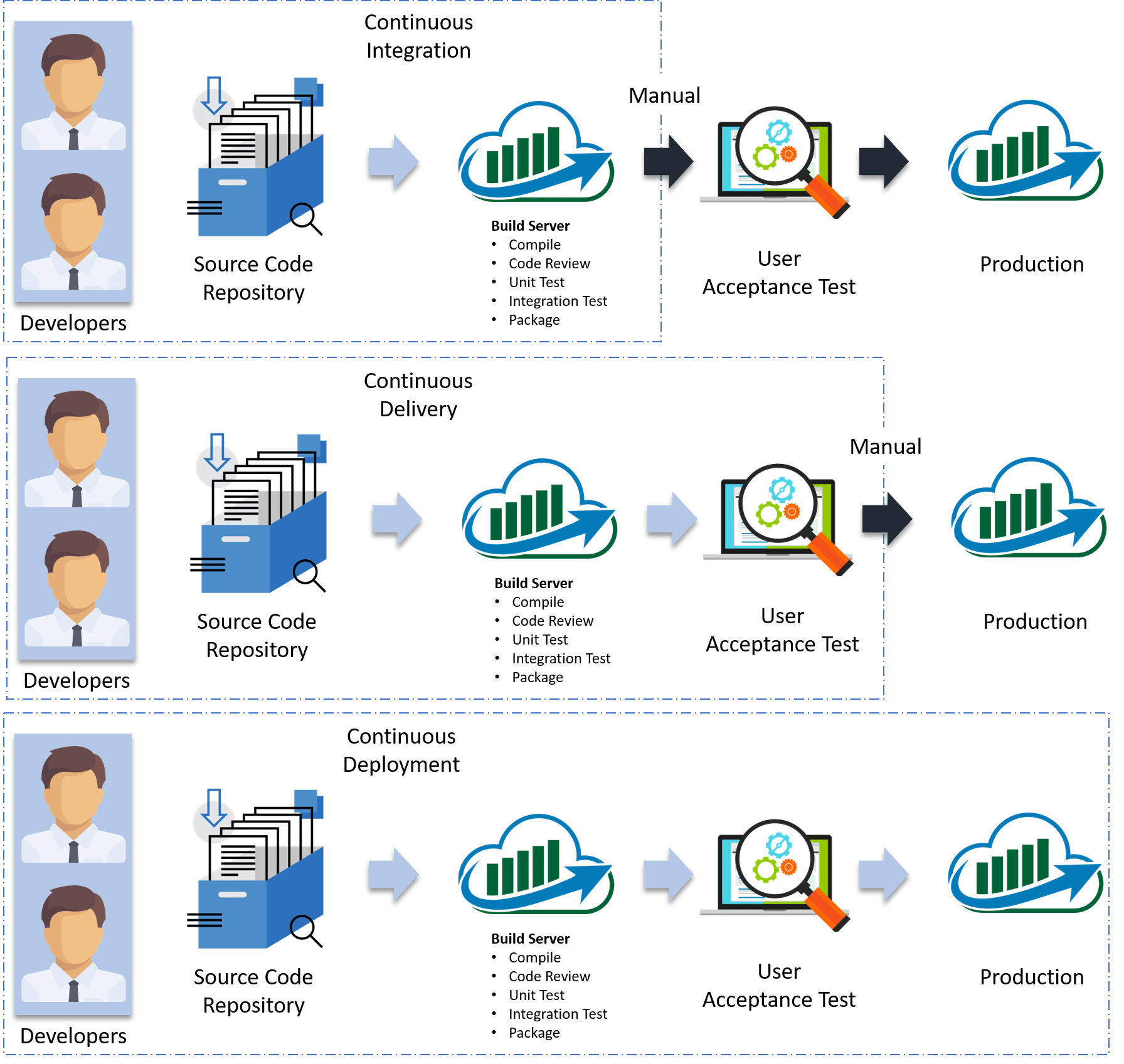
There are multiple tests that fall under this category. I will focus on a few, which are important for you to know, in order to understand this blog:

* **Functional/ Acceptance Testing:** It ensures that the specified functionality required in the system requirements works. It is done to make sure that the delivered product meets the requirements and works as the customer expected
* **System Testing:** It ensures that by putting the software in different environments (e.g., Operating Systems) it still works.
* **Stress Testing:** It evaluates how the system behaves under unfavorable conditions.
* **Beta Testing:** It is done by end users, a team outside development, or publicly releasing full pre-version of the product which is known as beta version. The aim of beta testing is to cover unexpected errors.

Now is the correct time for me to explain the difference between Continuous Integration, Delivery and Deployment.

## Differences Between Continuous Integration, Delivery And Deployment:

Visual content reaches an individual’s brain in a faster and more understandable way than textual information. So I am going to start with a diagram which clearly explains the difference:



In Continuous Integration, every code commit is build and tested, but, is not in a condition to be released. I mean the build application is not automatically deployed on the test servers in order to validate it using different types of Blackbox testing like – User Acceptance Testing (UAT).

In Continuous Delivery, the application is continuously deployed on the test servers for UAT. Or, you can say the application is ready to be released to production anytime. So, obviously Continuous Integration is necessary for Continuous Delivery.

Continuous Deployment is the next step past Continuous Delivery, where you are not just creating a deployable package, but you are actually deploying it in an automated fashion.

Let me summarize the differences using a table:

|  |  |  |
| --- | --- | --- |
| **Continuous Integration** | **Continuous Delivery** | **Continuous Deployment** |
| Automated build for every, commit | Automated build and UAT for every, commit | Automated build, UAT and release to production for every, commit |
| Independent of Continuous Delivery and Continuous Deployment | It is the next step after Continuous Integration | it is one step further Continuous Delivery |
| By the end, the application is not in a condition to be released to production | By the end, the application is in a condition to be released to the production. | The application is continuously deployed |
| Includes Whitebox testing | Includes Blackbox and Whitebox testing | It includes the entire process required to deploy the application |

In simple terms, Continuous Integration is a part of both Continuous Delivery and Continuous Deployment. And Continuous Deployment is like Continuous Delivery, except that releases happen automatically.

## Why We Need Continuous Delivery?

Let us understand this with an example.

Imagine there are 80 developers working on a large project. They are using Continuous Integration pipelines in order to facilitate automated builds. We know build includes Unit Testing as well. One day they decided to deploy the latest build that had passed the unit tests into a test environment.

This must be a lengthy but controlled approach to deployment that their environment specialists carried out. However, the system didn’t seem to work.

### What Might Be The Obvious Cause Of The Failure?

Well, the first reason that most of the people will think is that there is some problem with the configuration. Like most of the people even they thought so. They spent a lot of time trying to find what was wrong with the configuration of the environment, but they couldn’t find the problem.

### Why We Need Continuous Delivery - Continuous Delivery - Edureka

### One Perceptive Developer Took A Smart Approach:

Then one of the senior Developer tried the application on his development machine. It didn’t work there either.

He stepped back through earlier and earlier versions until he found that the system had stopped working three weeks earlier. A tiny, obscure bug had prevented the system from starting correctly. Although, the project had good unit test coverage. Despite this, 80 developers, who usually only ran the tests rather than the application itself, did not see the problem for three weeks.

### Problem Statement:

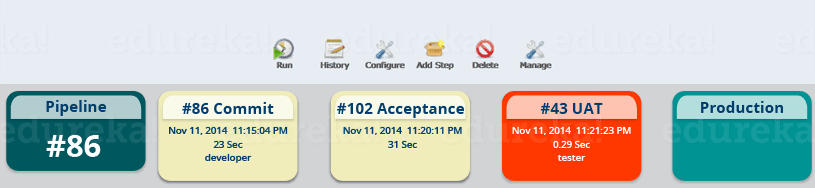
Without running Acceptance Tests in a production-like environment, they know nothing about whether the application meets the customer’s specifications, nor whether it can be deployed and survive in the real world. If they want timely feedback on these topics, they must extend the range of their continuous integration process.

Let me summarize the lessons learned by looking at the above problems:

* Unit Tests only test a developer’s perspective of the solution to a problem. They have only a limited ability to prove that the application does what it is supposed to from a users perspective. They are not enough to identify the real functional problems.
* Deploying the application on the test environment is a complex, manually intensive process that was quite prone to error. This meant that every attempt at deployment was a new experiment — a manual, error-prone process.

## Solution – Continuous Delivery Pipeline (Automated Acceptance Test):

They took Continuous Integration (Continuous Delivery) to the next step and introduced a couple of simple, automated Acceptance Tests that proved that the application ran and could perform its most fundamental function. The majority of the tests running during the Acceptance Test stage are Functional Acceptance Tests.



Basically, they built a Continuous Delivery pipeline, in order to make sure that the application is seamlessly deployed on the production environment, by making sure that the application works fine when deployed on the test server which is a replica of the production server.

*Enough of the theory, I will now show you how to create a Continuous Delivery pipeline using Jenkins.*

## Continuous Delivery Pipeline Using Jenkins:

Here I will be using Jenkins to create a Continuous Delivery Pipeline, which will include the following tasks:

### Steps involved in the Demo:

* Fetching the code from GitHub
* Compiling the source code
* Unit testing and generating the JUnit test reports
* Packaging the application into a WAR file and deploying it on the Tomcat server



Pre-requisites:

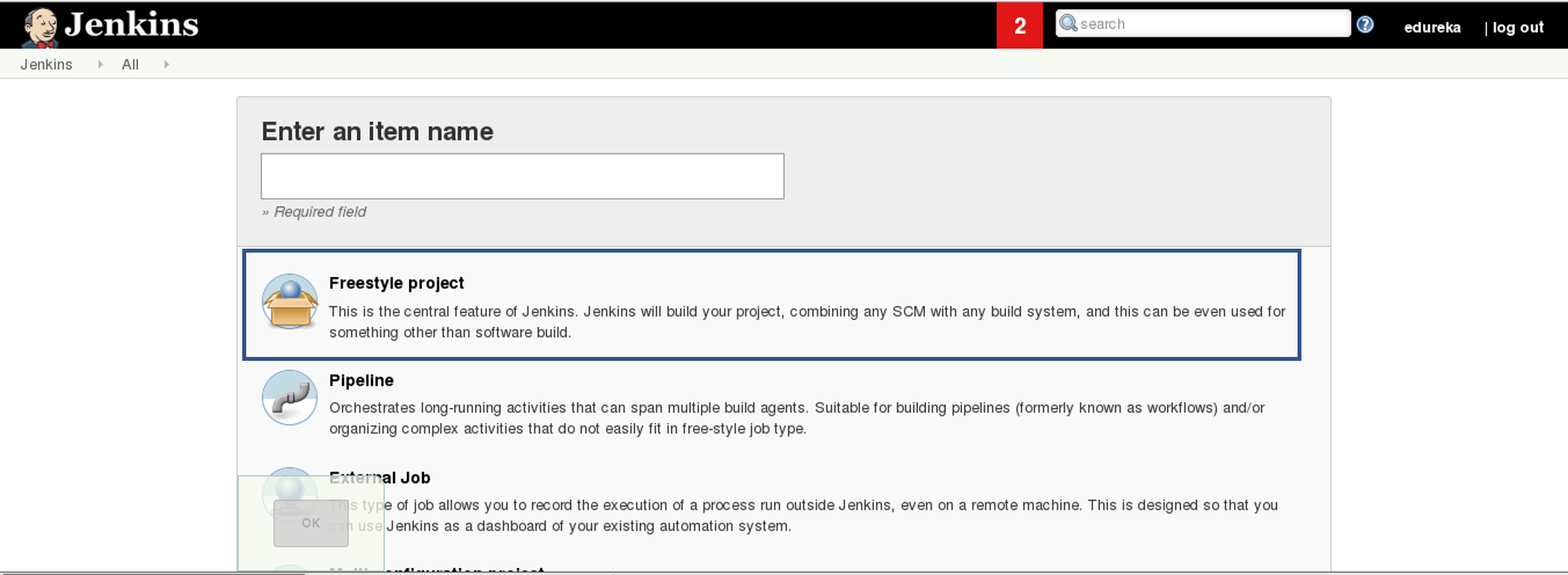
* CentOS 7 Machine
* Jenkins 2.121.1
* Docker
* Tomcat 7

### Step – 1 Compiling The Source Code:

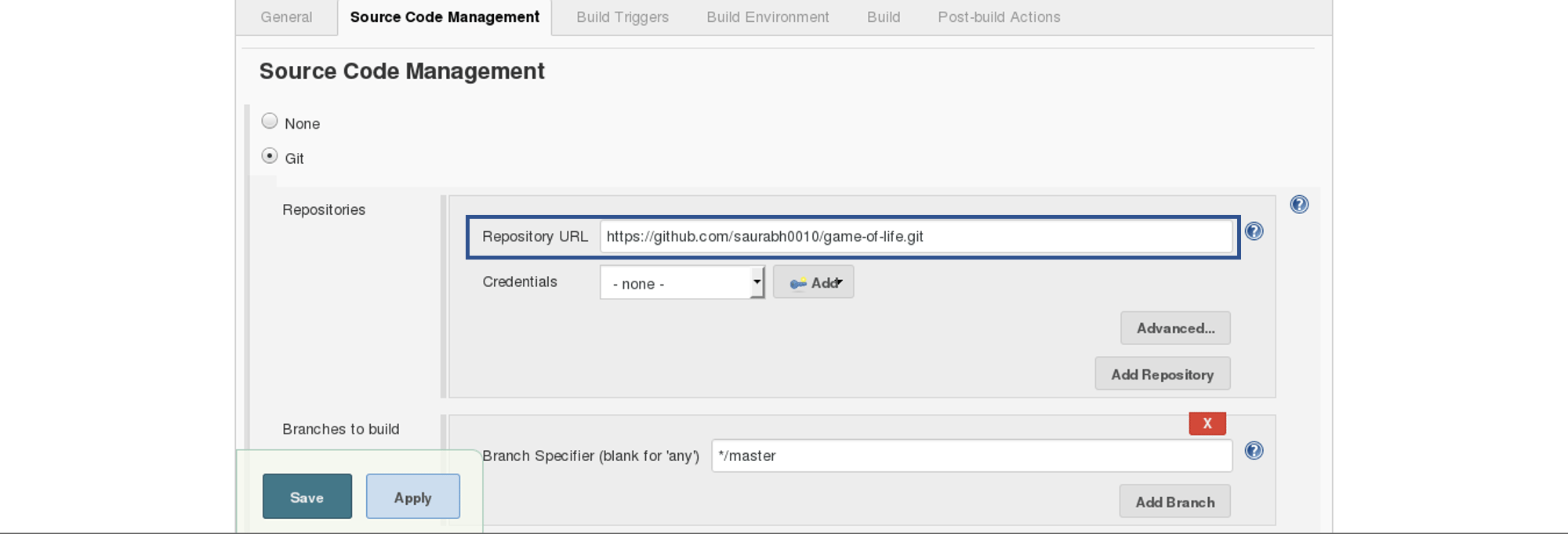
Let’s begin by first creating a Freestyle project in Jenkins. Consider the below screenshot:



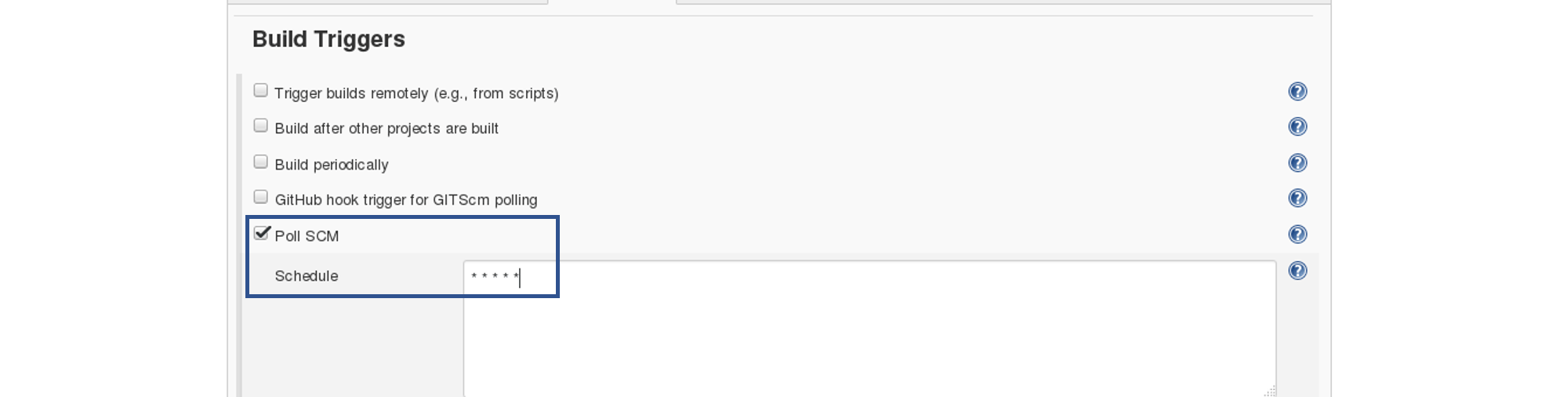
Give a name to your project and select Freestyle Project:



When you scroll down you will find an option to add source code repository, select git and add the repository URL, in that repository there is a pom.xml fine which we will use to build our project. Consider the below screenshot:



Now we will add a Build Trigger. Pick the poll SCM option, basically, we will configure Jenkins to poll the GitHub repository after every 5 minutes for changes in the code. Consider the below screenshot:



Before I proceed, let me give you a small introduction to the Maven Build Cycle.

Each of the build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

Following is the list of build phases:

* validate – validate the project is correct and all necessary information is available
* compile – compile the source code of the project
* test – test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package – take the compiled code and package it in its distributable format, such as a JAR.
* verify – run any checks on results of integration tests to ensure quality criteria are met
* install – install the package into the local repository, for use as a dependency in other projects locally
* deploy – done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

I can run the below command, for compiling the source code, unit testing and even packaging the application in a war file:

mvn clean package

You can also break down your build job into a number of build steps. This makes it easier to organize builds in clean, separate stages.

So we will begin by compiling the source code. In the build tab, click on invoke top level maven targets and type the below command:

compile

Consider the below screenshot:

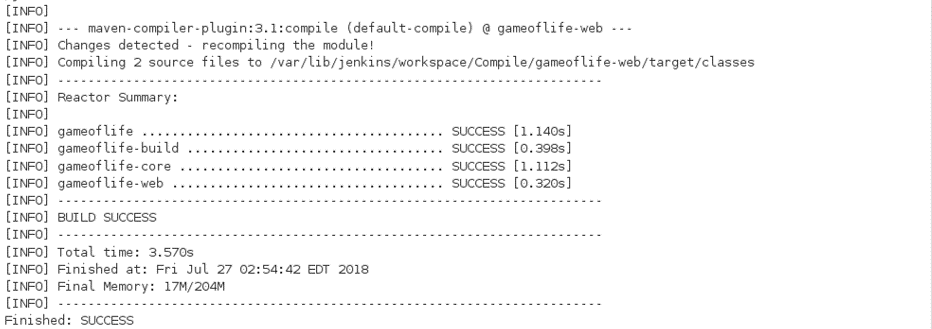


This will pull the source code from the GitHub repository and will also compile it (Maven Compile Phase).

Click on Save and run the project.



Now, click on the console output to see the result.



**Step – 2 Unit Testing:**

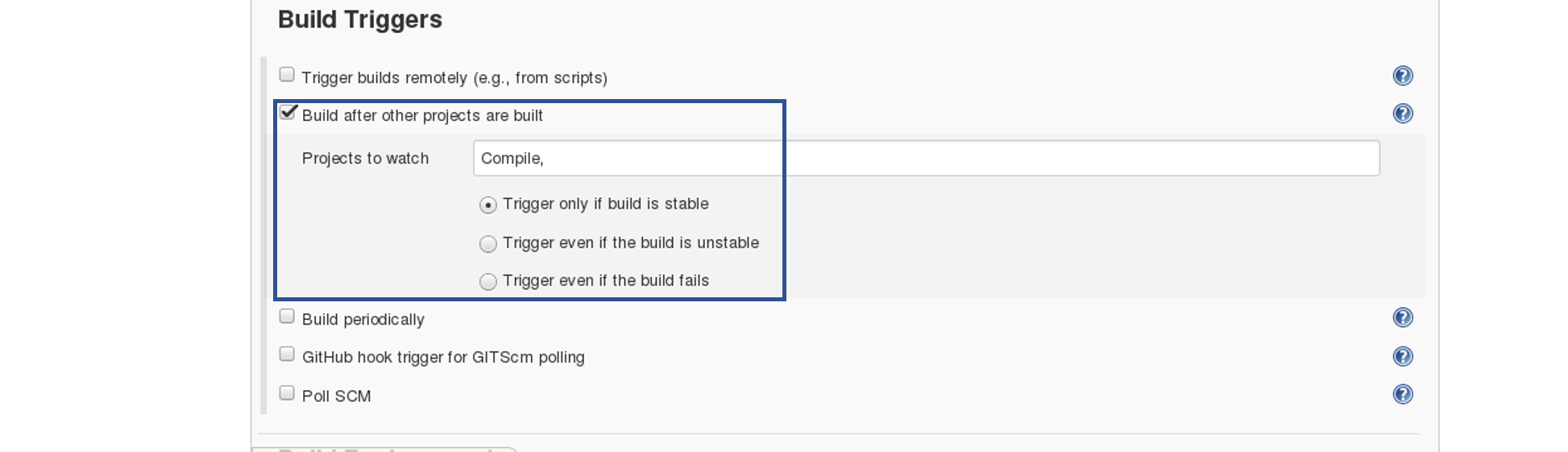
Now we will create one more Freestyle Project for unit testing.

Add the same repository URL in the source code management tab, like we did in the previous job.

Now, in the “Buid Trigger” tab click on the “build after other projects are built”. There type the name of the previous project where we are compiling the source code, and you can select any of the below options:

* Trigger only if the build is stable
* Trigger even if the build is unstable
* Trigger even if the build fails

I think the above options are pretty self-explanatory so, select any one. Consider the below screenshot:



In the Build tab, click on invoke top level maven targets and use the below command:

test

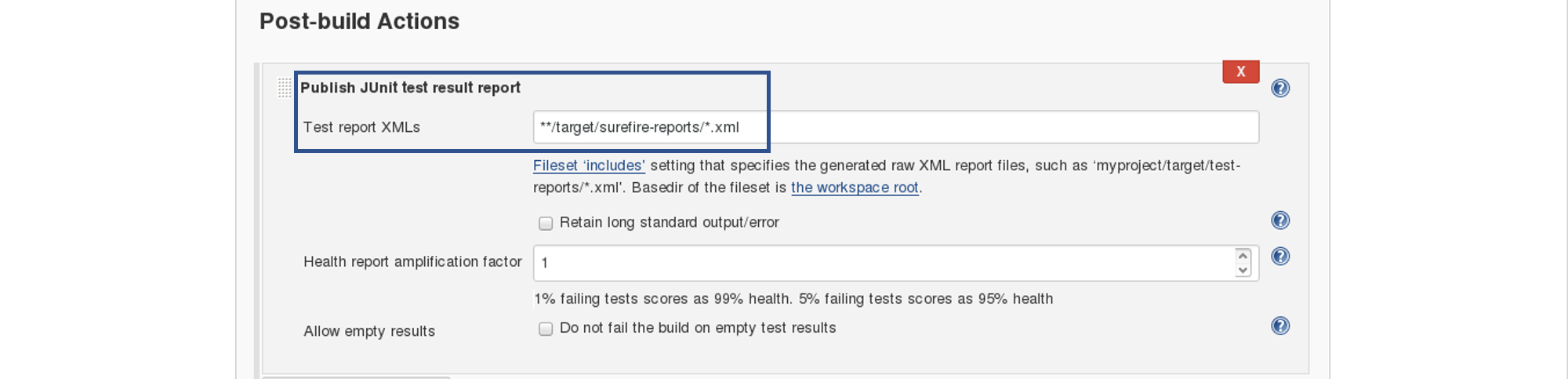
Jenkins also does a great job of helping you display your test results and test result trends.

The de facto standard for test reporting in the Java world is an XML format used by JUnit. This format is also used by many other Java testing tools, such as TestNG, Spock, and Easyb. Jenkins understands this format, so if your build produces JUnit XML test results, Jenkins can generate nice graphical test reports and statistics on test results over time, and also let you view the details of any test failures. Jenkins also keeps track of how long your tests take to run, both globally, and per test—this can come in handy if you need to track down performance issues.

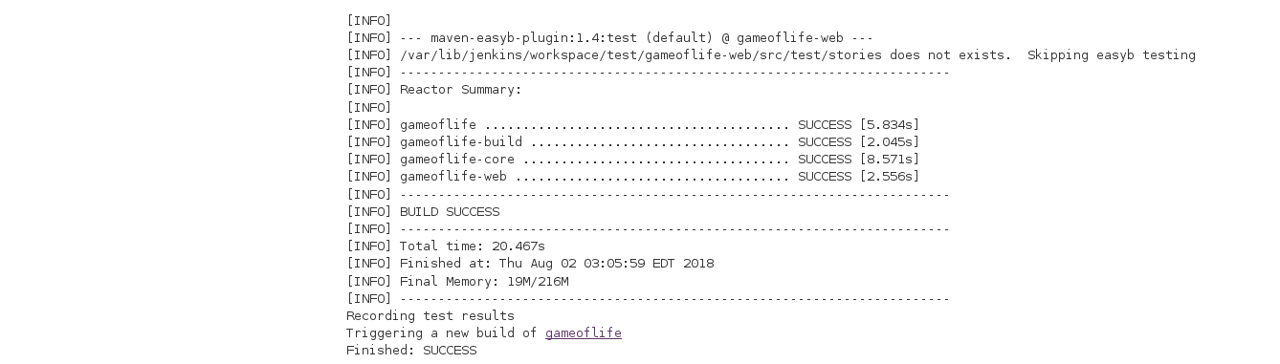
So the next thing we need to do is to get Jenkins to keep tabs on our unit tests.

Go to the Post-build Actions section and tick “Publish JUnit test result report” checkbox. When Maven runs unit tests in a project, it automatically generates the XML test reports in a directory called surefire-reports. So enter “\*\*/target/surefire-reports/\*.xml” in the “Test report XMLs” field. The two asterisks at the start of the path (“\*\*”) are a best practice to make the configuration a bit more robust: they allow Jenkins to find the target directory no matter how we have configured Jenkins to check out the source code.

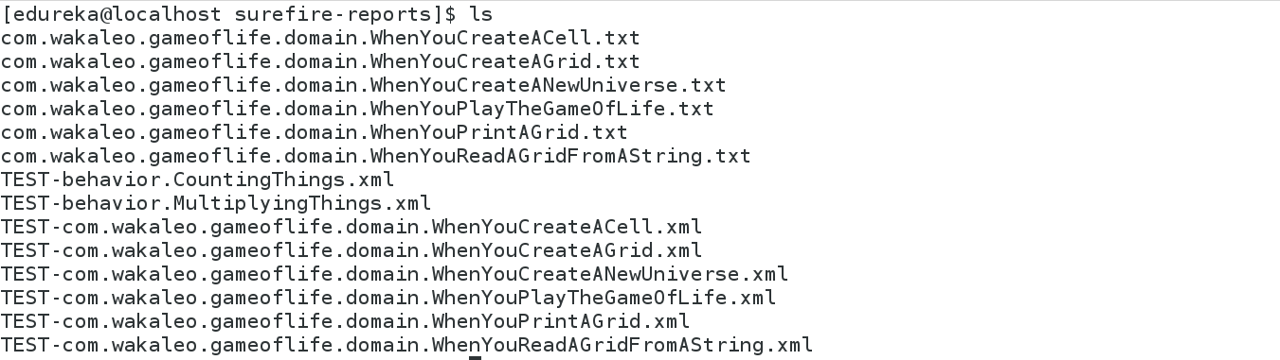
\*\*/target/surefire-reports/\*.xml



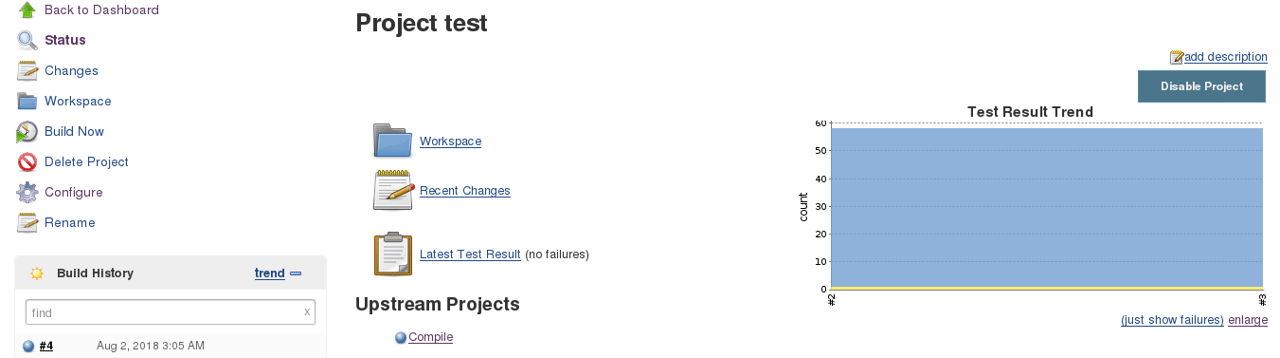
Again save it and click on Build Now.

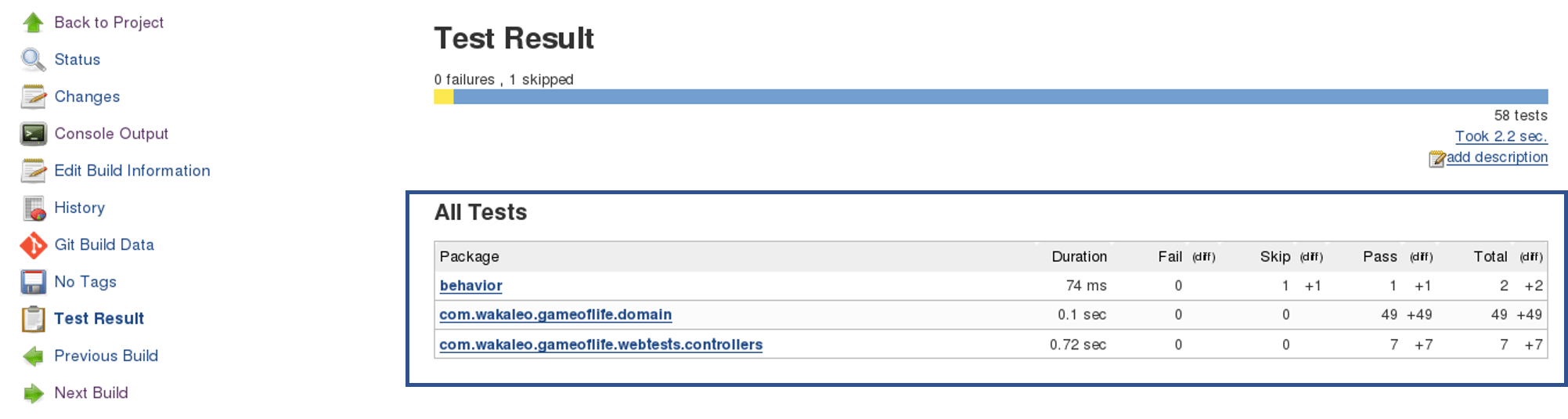


Now, the JUnit report is written to /var/lib/jenkins/workspace/test/gameoflife-core/target/surefire-reports/TEST-behavior.



In the Jenkins dashboard you can also notice the test results:



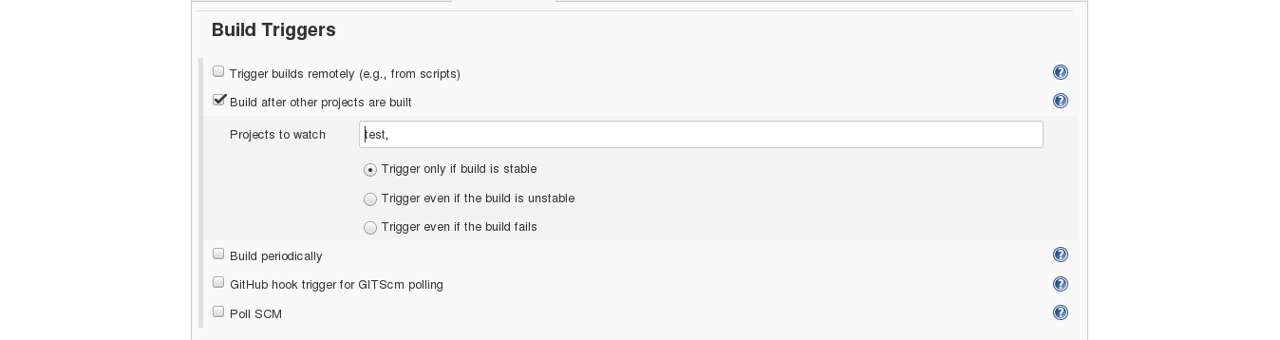


**Step – 3 Creating A WAR File And Deploying On The Tomcat Server:**

Now, the next step is to package our application in a WAR file and deploy that on the Tomcat server for User Acceptance test.

Create one more freestyle project and add the source code repository URL.

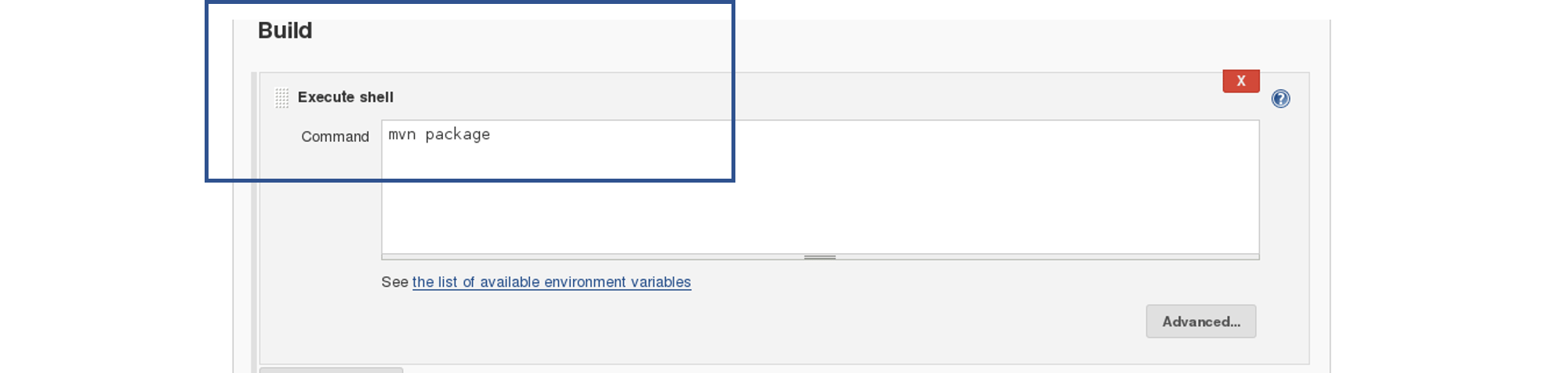
Then in the build trigger tab, select build when other projects are built, consider the below screenshot:



Basically, after the test job, the deployment phase will start automatically.

In the build tab, select shell script. Type the below command to package the application in a WAR file:

mvn package

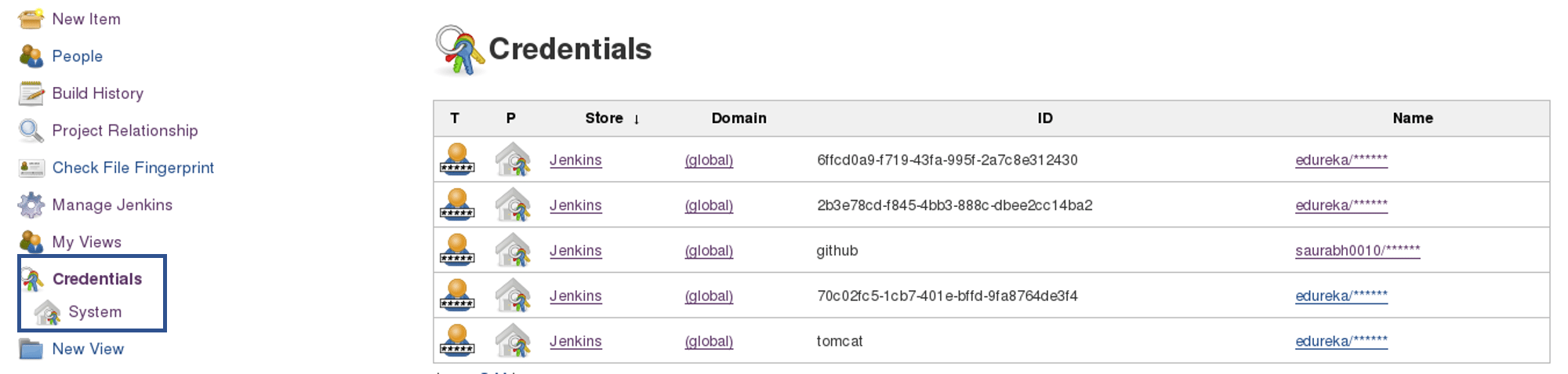


Next step is to deploy this WAR file to the Tomcat server. In the “Post-Build Actions” tab select deploy war/ear to a container. Here, give the path to the war file and give the context path. Consider the below screenshot:

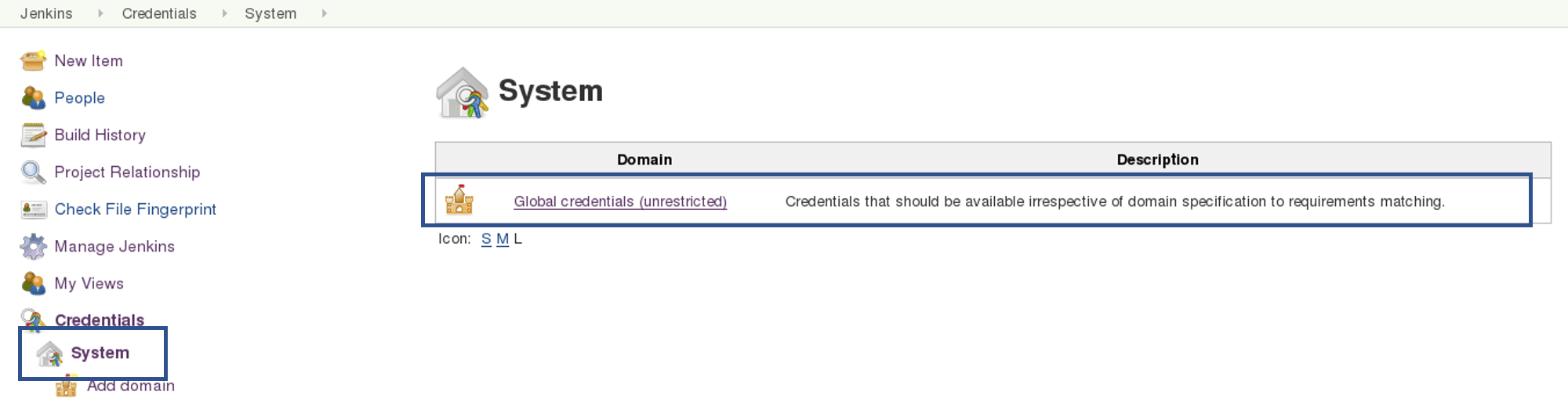


Select the Tomcat credentials and, notice the above screenshot. Also, you need to give the URL of your Tomcat server.

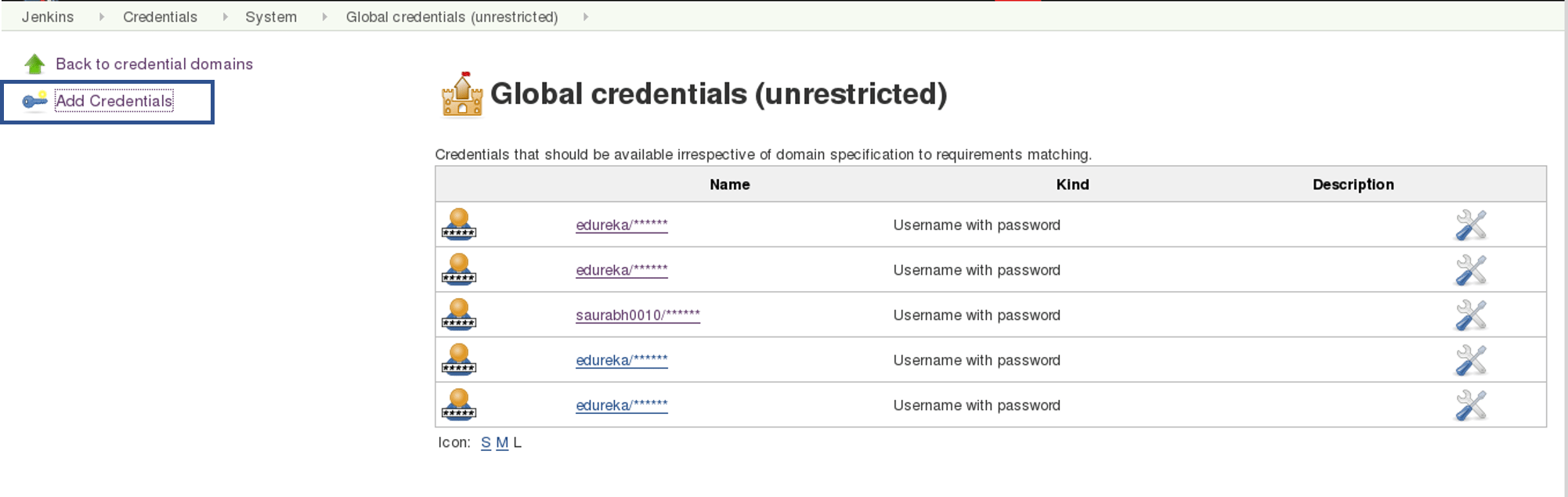
In order to add credentials in Jenkins, click on credentials option on the Jenkins dashboard.



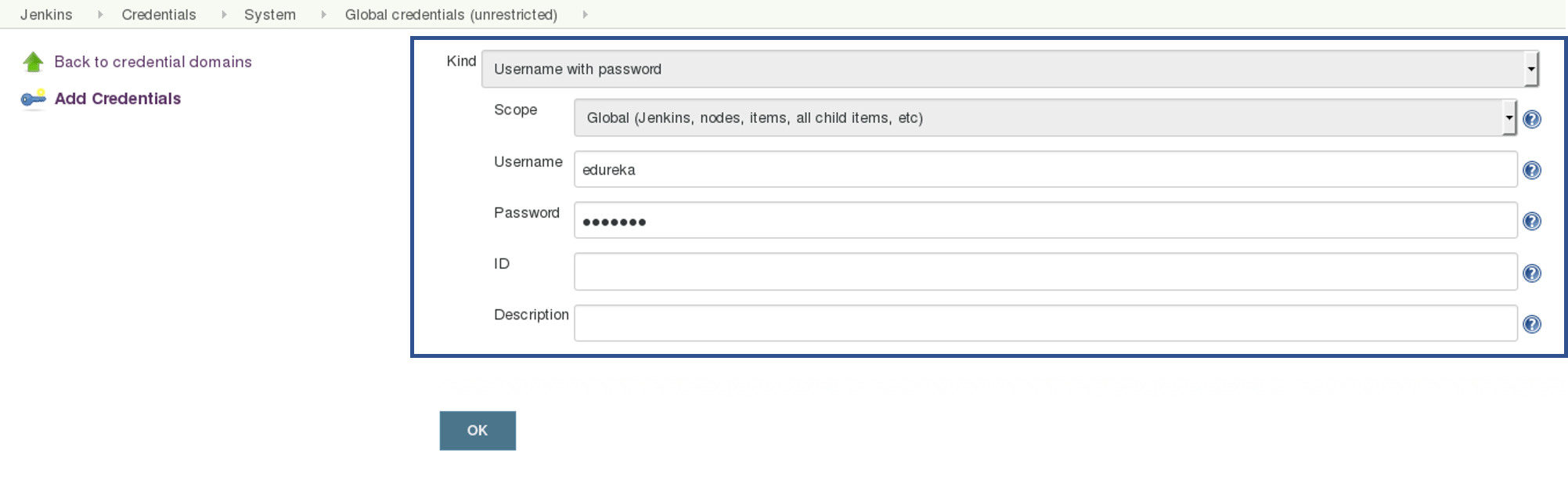
Click on System and select global credentials.



Then you will find an option to add the credentials. Click on it and add credentials.



Add the Tomcat credentials, consider the below screenshot.

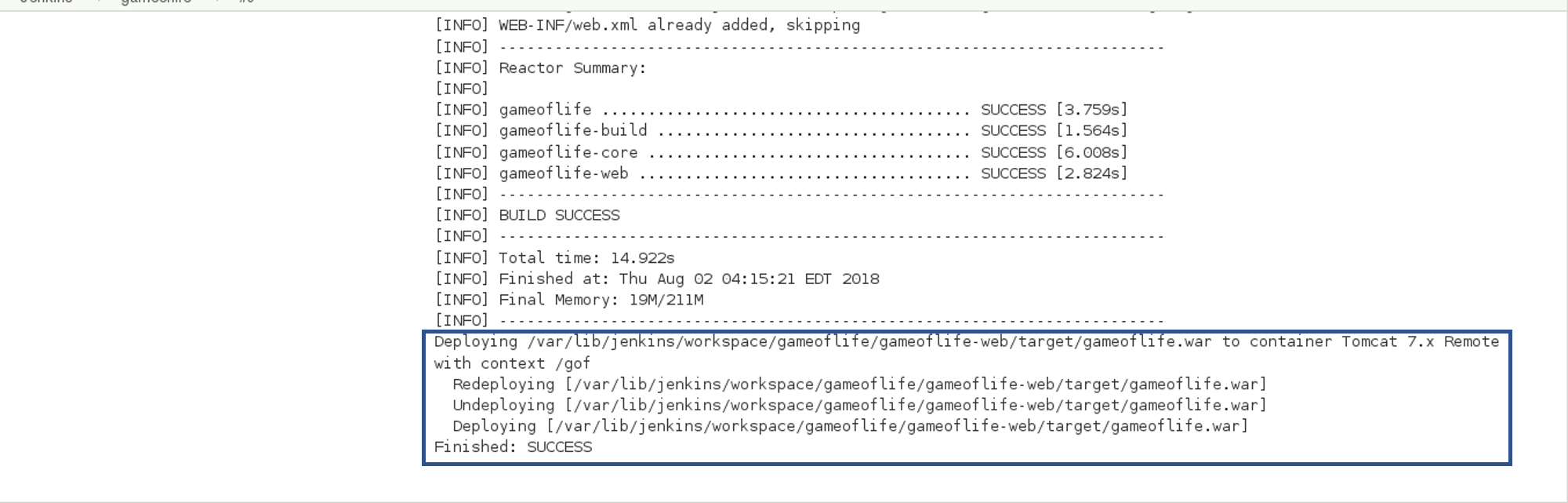


Click on OK.

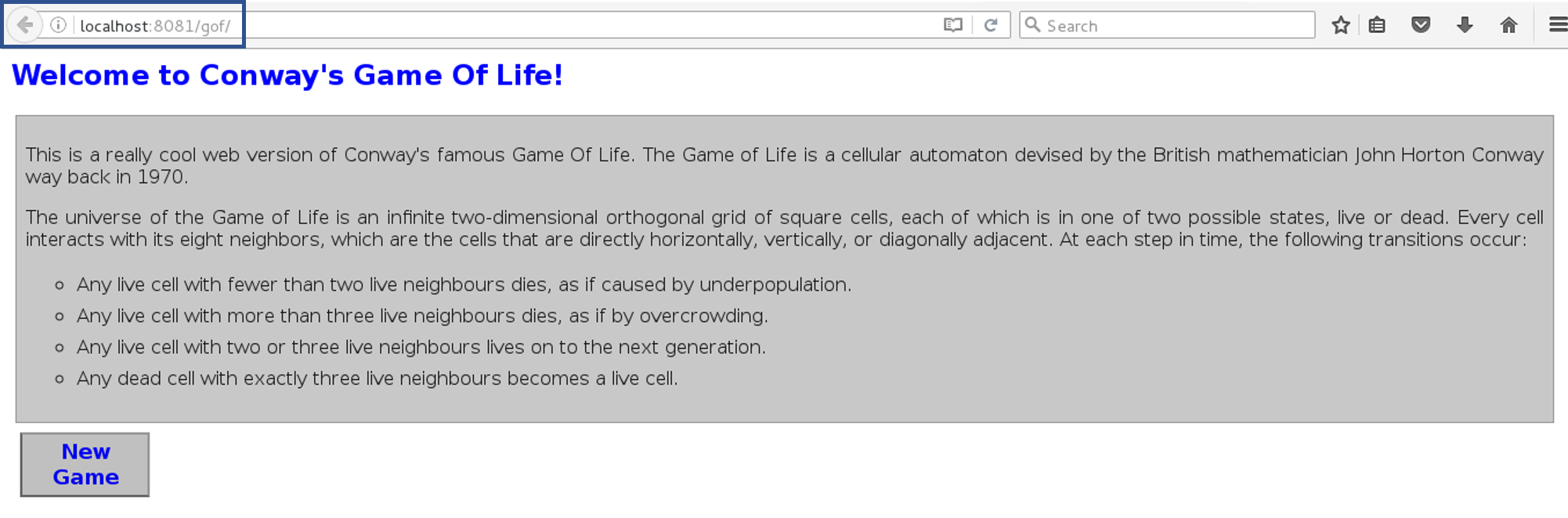
Now in your Project Configuration, add the tomcat credentials which you have inserted in the previous step.



Click on Save and then select Build Now.



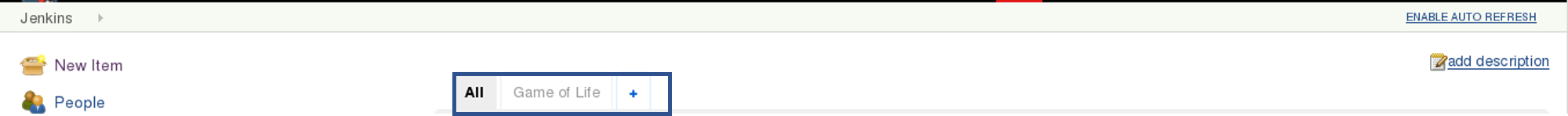
Go to your tomcat URL, with the context path, in my case it is http://localhost:8081. Now add the context path in the end, consider the below Screenshot:



Link - http://localhost:8081/gof

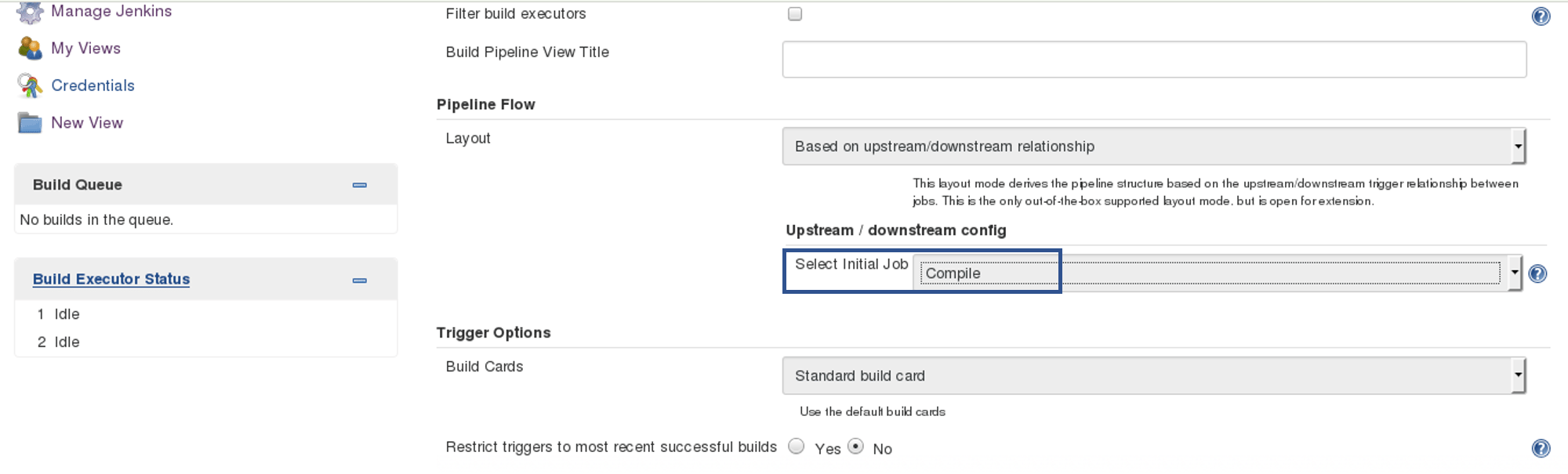
I hope you have understood the meaning of the context path.

Now create a pipeline view, consider the below screenshot:

**

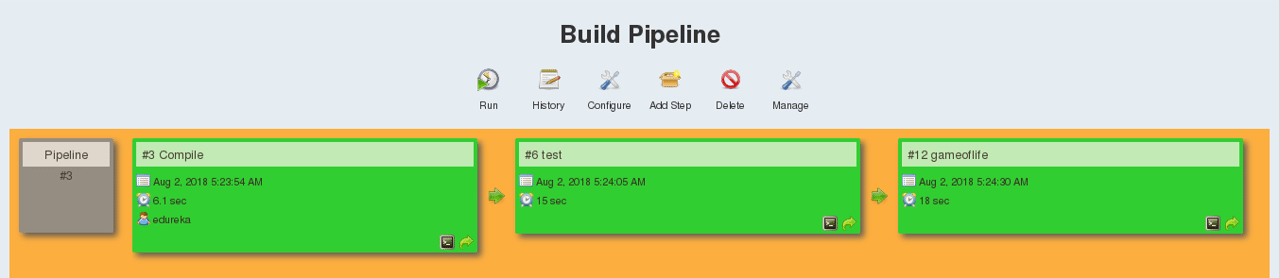
Click on the plus icon, to create a new view.

Configure the pipeline the way you want, consider the below screenshot:



I did not change anything apart from selecting the initial job. So my pipeline will start from compile. Based on the way I have configured other jobs, after compile testing and deployment will happen.

Finally, you can test the pipeline by clicking on RUN. After every five minutes, if there is a change in the source code, the entire pipeline will be executed.

**

So we are able to continuously deploy our application on the test server for user acceptance test (UAT).

CI CD Pipeline: Learn How to Setup a CI CD Pipeline From Scratch

CI CD Pipeline implementation or the Continuous Integration/Continuous Deployment software is the backbone of the modern DevOps environment. You can find the requirement of [*Continuous Integration*](https://www.edureka.co/blog/continuous-integration/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)*&*[*Continuous Deployment*](https://www.edureka.co/blog/continuous-deployment/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)*skills* in various job roles such as [Data Engineer](https://www.edureka.co/blog/big-data-engineer-skills?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline), Cloud Architect, [Data Scientist](https://www.edureka.co/blog/data-scientist-salary?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline), etc. CI/CD bridges the gap between development and operations teams by automating build, test, and deployment of applications. In this blog, we will know What is CI CD pipeline and how it works.

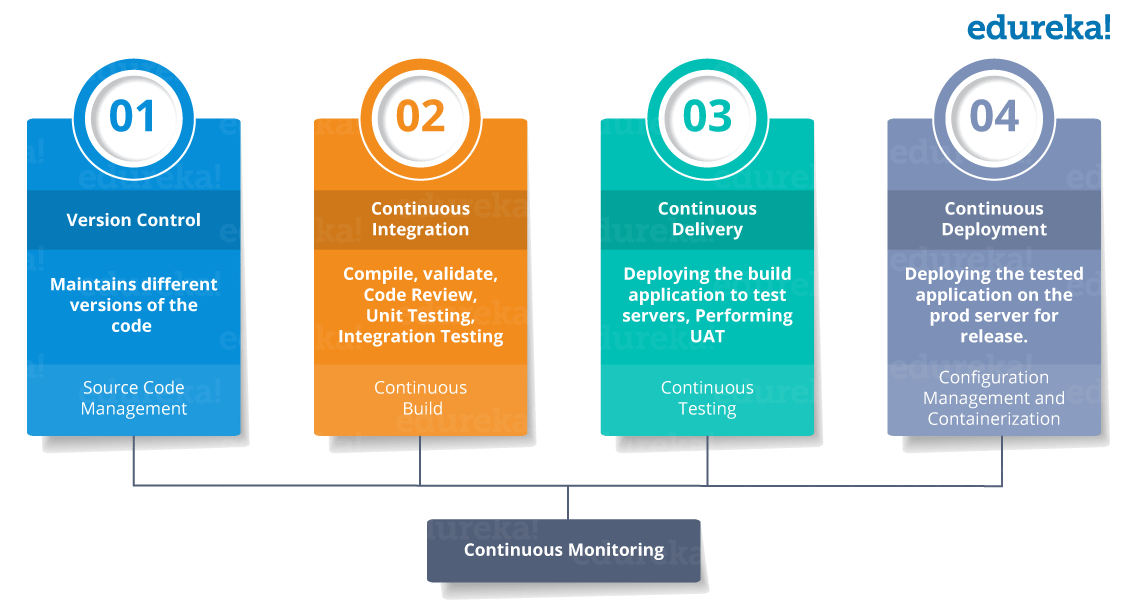
Before moving onto the CI CD pipeline’s working, let’s start by understanding DevOps.

[**What is DevOps?**](https://www.edureka.co/blog/what-is-devops/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)

DevOps is a software development approach which involves continuous development, continuous testing, [continuous integration](https://www.edureka.co/blog/continuous-integration?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline), [continuous deployment](https://www.edureka.co/blog/continuous-deployment/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline) and continuous monitoring of the software throughout its development life cycle. This is exactly the process adopted by all the top companies to develop high-quality software and shorter development life cycles, resulting in greater customer satisfaction, something that every company wants.

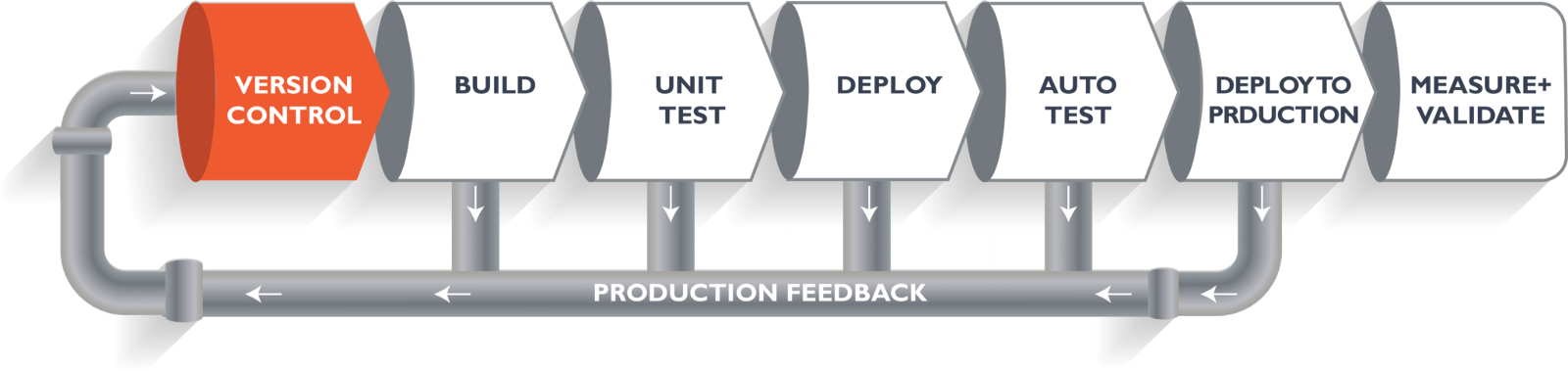
**DevOps Stages**

Your understanding of what is DevOps is incomplete without learning about its life cycle. Let us now look at the DevOps lifecycle and explore how they are related to the software development stages.



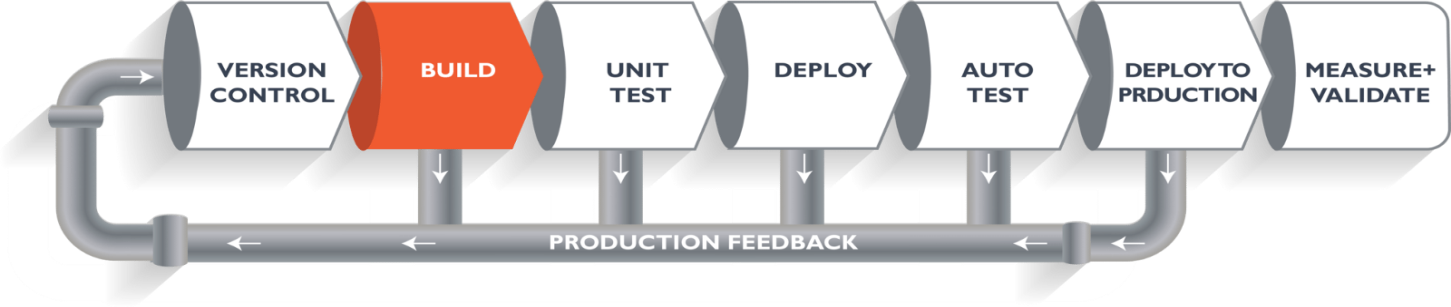
**What is CI CD Pipeline?**

CI stands for [Continuous Integration](https://www.edureka.co/blog/continuous-integration/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)and CD stands for [Continuous Delivery](https://www.edureka.co/blog/continuous-delivery/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)and [Continuous Deployment](https://www.edureka.co/blog/continuous-deployment/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline). You can think of it as a process which is similar to a software development lifecycle.  
 Now let us see how does it work.



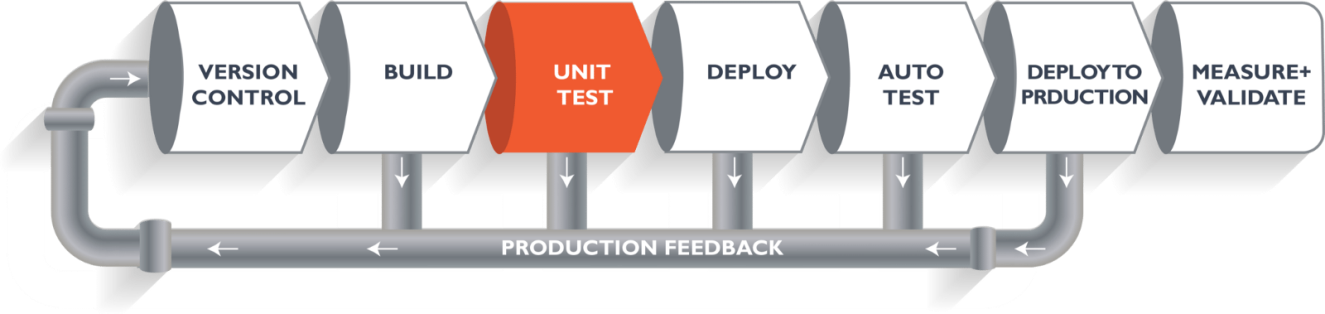
The above pipeline is a logical demonstration of how software will move along the various phases or stages in this lifecycle, before it is delivered to the customer or before it is live on production.

Let’s take a scenario of CI CD Pipeline. Imagine you’re going to build a web application which is going to be deployed on live web servers. You will have a set of developers who are responsible for writing the code which will further go on and build the web application. Now, when this code is committed into a version control system(such as [git](https://www.edureka.co/blog/git-tutorial/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline" \t "_blank), svn) by the team of developers. Next, it goes through the **build phase** which is the first phase of the pipeline, where developers put in their code and then again code goes to the version control system having a proper version tag.



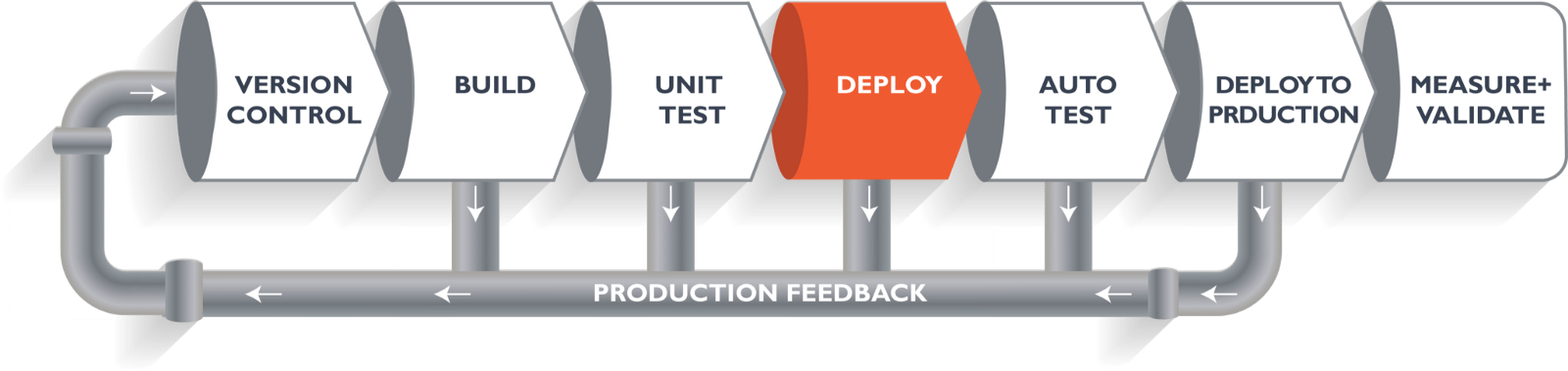
Suppose we have a [Java](https://www.edureka.co/blog/java-tutorial/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline) code and it needs to be compiled before execution. So, through the version control phase, it again goes to build phase where it gets compiled. You get all the features of that code from various branches of the repository, which merge them and finally use a compiler to compile it. This whole process is called the **build phase**.

**Testing** Phase:



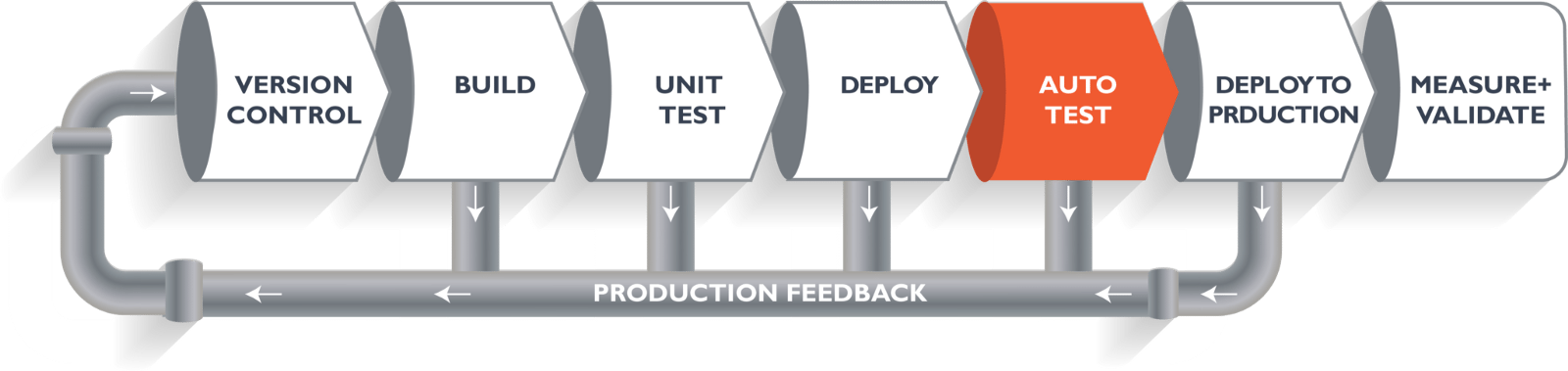
Once the build phase is over, then you move on to the **testing phase**. In this phase, we have various kinds of testing, one of them is the *unit test* (where you test the chunk/unit of software or for its sanity test).

**Deploy Phase:**



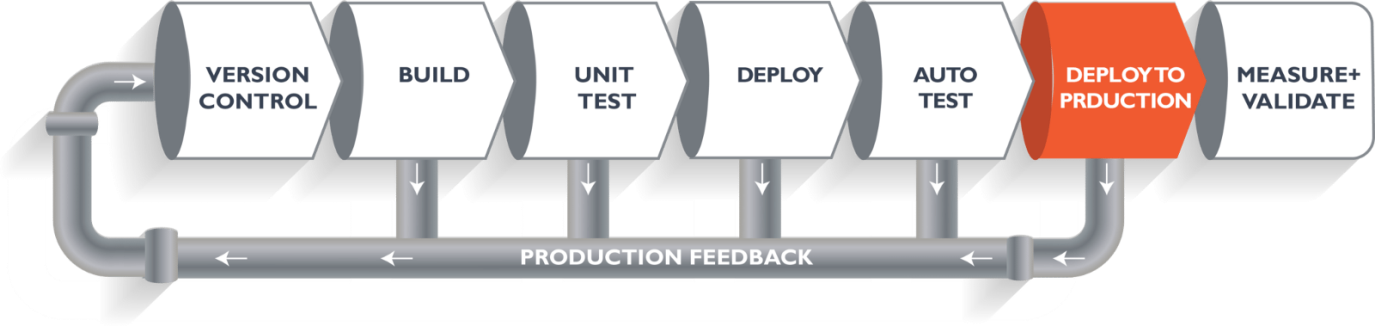
When the test is completed, you move on to the **deploy phase**, where you deploy it into a staging or a test server. Here, you can view the code or you can view the app in a simulator.

**Auto Test Phase:**



Once the code is deployed successfully, you can run another set of a sanity test. If everything is accepted, then it can be deployed to production.

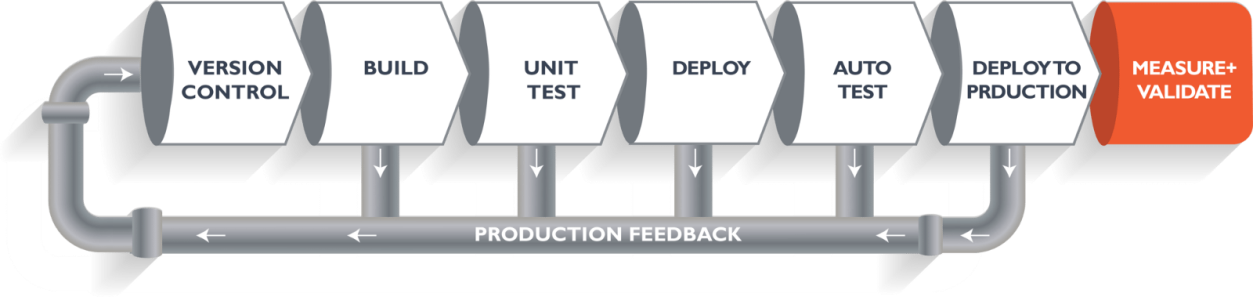
**Deploy to Production:**



Meanwhile in every step, if there is some error, you can shoot a mail back to the development team so that they can fix them. Then they will push it into the version control system and goes back into the pipeline.

Once again if there is any error reported during testing, again the feedback goes to the dev team where they fix it and the process re-iterates if required.

**Measure+Validate:**



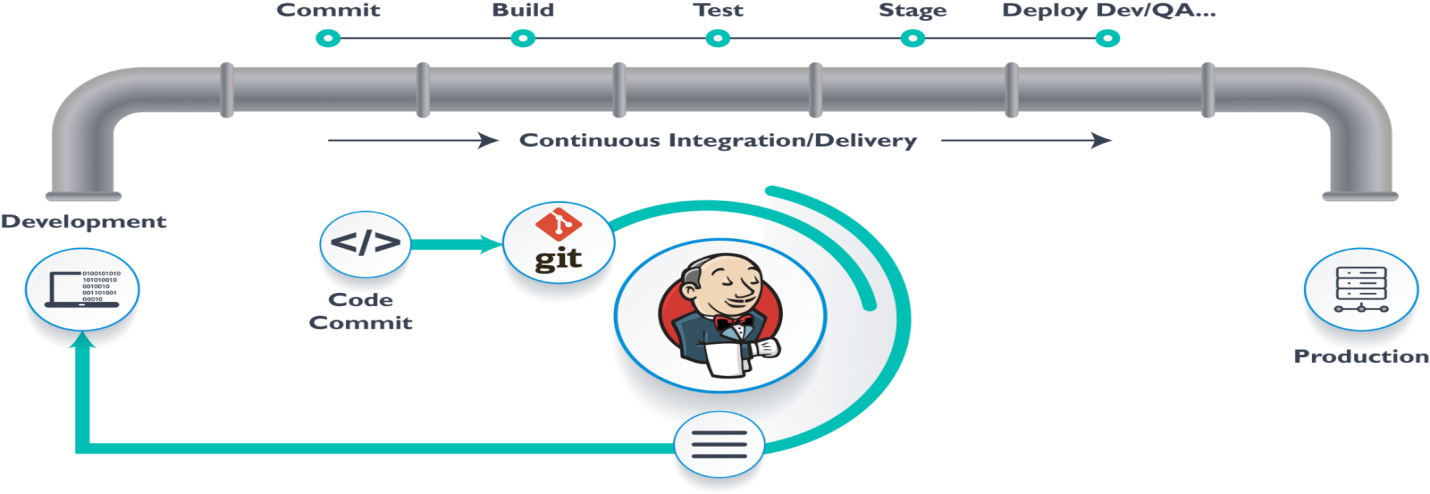
So, this lifecycle continues until we get a code or a product which can be deployed in the production server where we measure and validate the code.

We have understood CI CD Pipeline and its working, now we will move on to understand what Jenkins is and how we can deploy the demonstrated code using Jenkins and automate the entire process.

[**Jenkins**](https://www.edureka.co/blog/what-is-jenkins/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)**— The Ultimate CI Tool and Its Importance in CI CD Pipeline**

Our task is to automate the entire process, from the time the development team gives us the code and commits it to the time we get it into production.

Our task is to automate the pipeline in order to make the entire software development lifecycle on the dev-ops mode/ automated mode. For this, they would need automation tools.



[*Jenkins*](https://www.edureka.co/blog/jenkins-tutorial?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline) provides us with various interfaces and tools in order to automate the entire process.

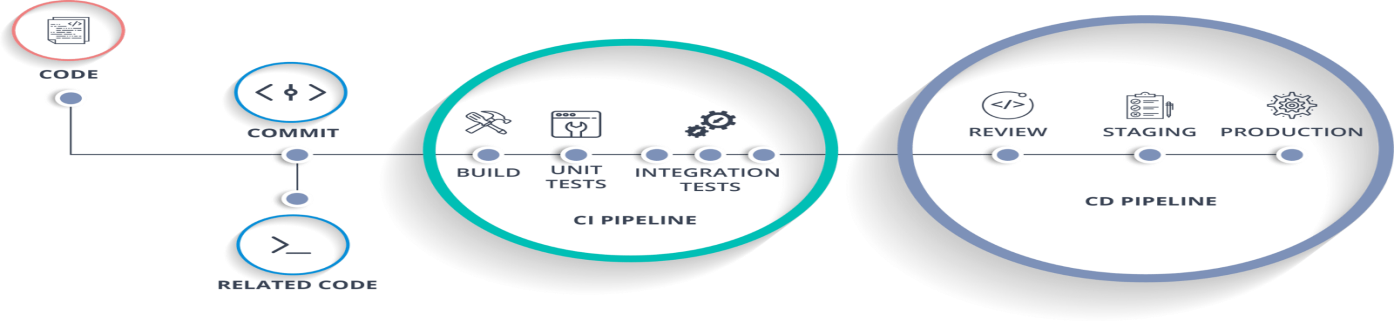
So what happens, we have a git repository where the development team will commit the code. Then Jenkins takes over from there which is a front-end tool where you can define your entire job or the task. Our job is to ensure the continuous integration and delivery process for that particular tool or for the particular application.

From Git, Jenkins pulls the code and then moves it to the **commit phase**, where the code is committed from every branch. Then Jenkins moves it into the **build phase** where we compile the code. If it is Java code, we use tools like maven in Jenkins and then compile that code, which we can be deployed to run a series of tests. These test cases are overseen by Jenkins again.

Then it moves on to the staging server to deploy it using **[Docker](https://www.edureka.co/blog/docker-tutorial?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline" \t "_blank)**. After a series of Unit Tests or sanity test, it moves to the production.

This is how the delivery phase is taken care by a tool called **Jenkins,** which automate everything. Now in order to deploy it, we will need an environment which will replicate the production environment, I.e., **Docker**.

### [Docker](https://www.edureka.co/blog/docker-tutorial?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline)



[*Docker*](https://www.edureka.co/blog/docker-explained/?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline) is just like a virtual environment in which we can create a server. It takes a few seconds to create an entire server and deploy the artifacts which we want to test. But here the question arises,

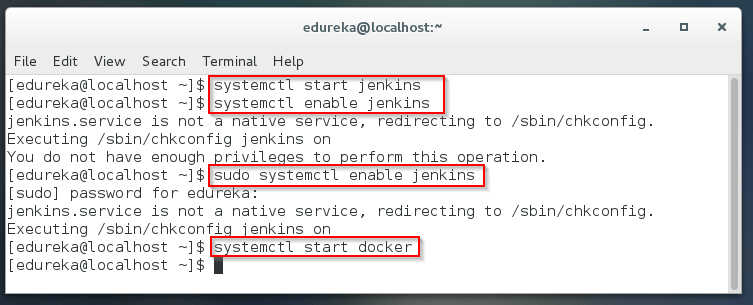
*Why do we use docker?*

As said earlier, you can run the entire cluster in a few seconds. We have storage registry for images where you build your image and store it forever. You can use it anytime in any environment which can replicate itself.

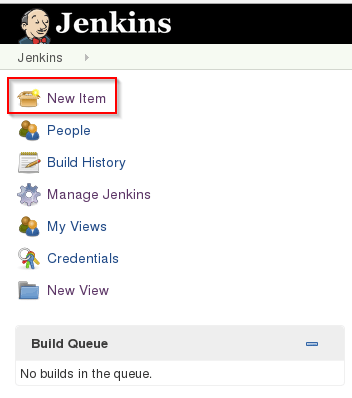
### Hands-On: Building CI CD Pipeline Using Docker and Jenkins

**Step 1:** Open your terminal in your VM. Start Jenkins and Docker using the commands “**systemctl start jenkins**“, “**systemctl enable jenkins**“, “**systemctl start docker**“.

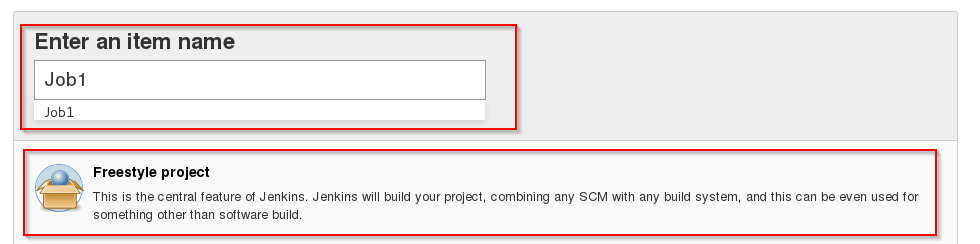
**Note:** Use **sudo**before the commands if it display “privileges error”.



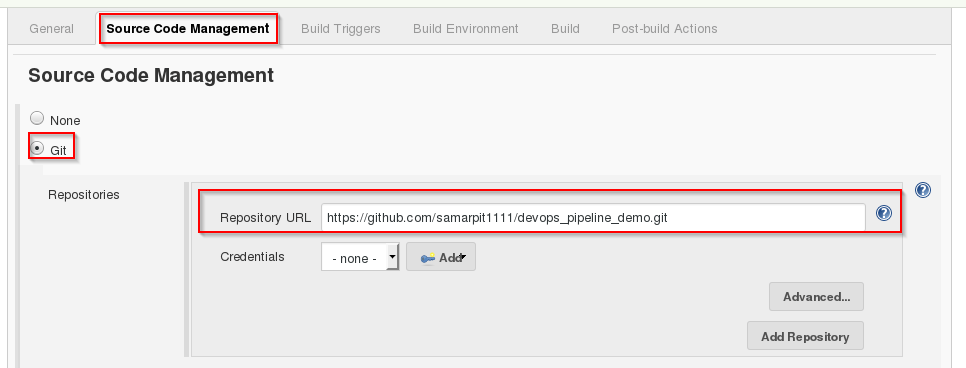
**Step 2:** Open your Jenkins on your specified port. Click on **New Item**to create a Job.



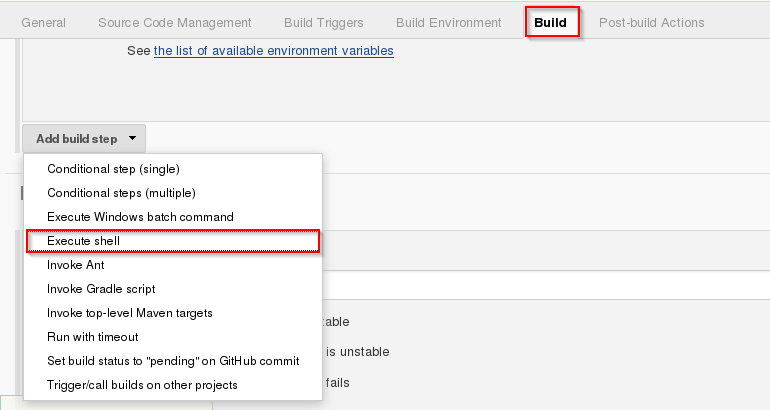
**Step 3:** Select **freestyle** project and provide the item name (here I have given Job1) and click OK.



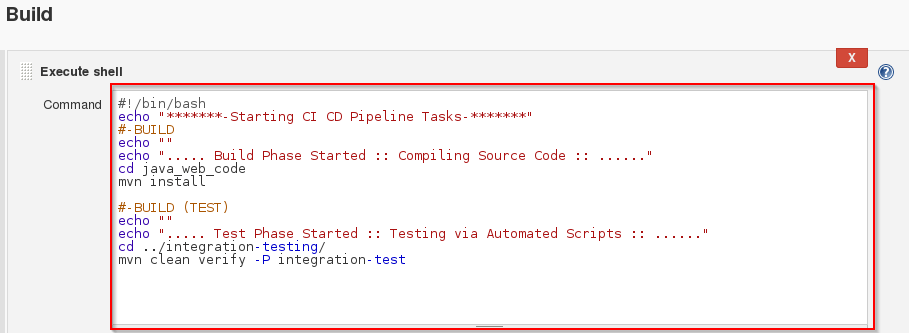
**Step 4:** Select **Source Code Management** and provide the **Git** repository. Click on **Apply** and **Save** button.



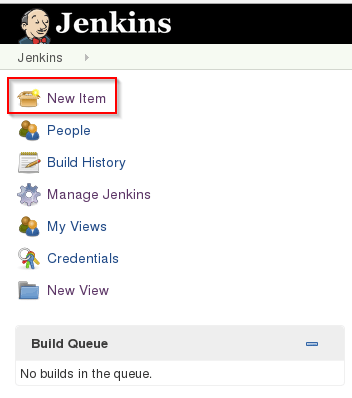
**Step 5:** Then click on **Build->Select Execute Shell**.



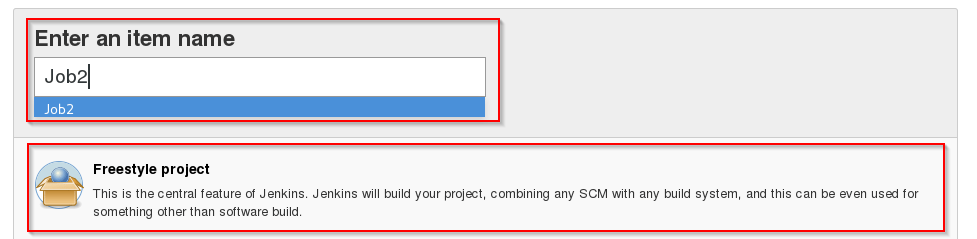
**Step 6:** Provide the shell commands. Here it will build the archive file to get a war file. After that, it will get the code which is already pulled and then it uses maven to install the package. So, it simply installs the dependencies and compiles the application.



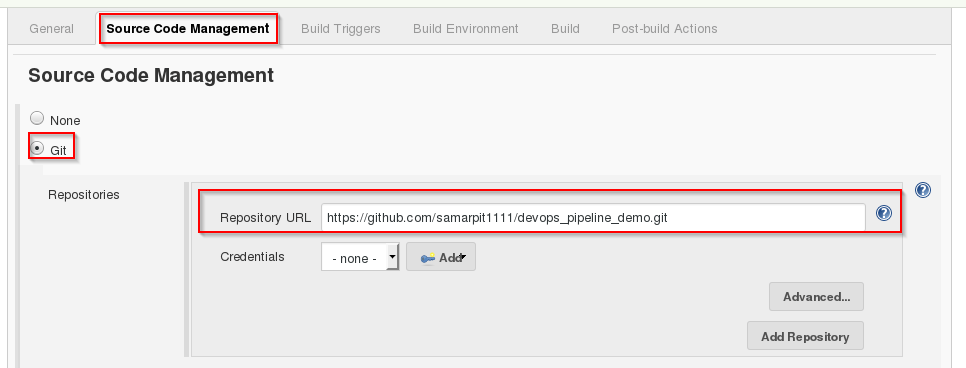
**Step 7:** Create the new **Job** by clicking on New Item.



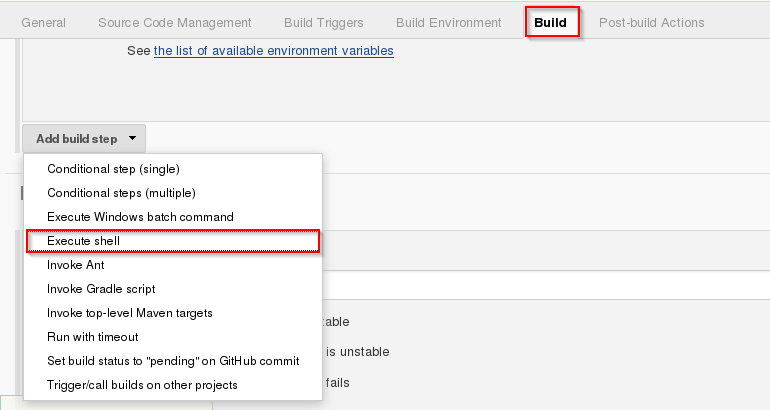
**Step 8:** Select **freestyle** project and provide the item name (here I have given Job2) and click on OK.



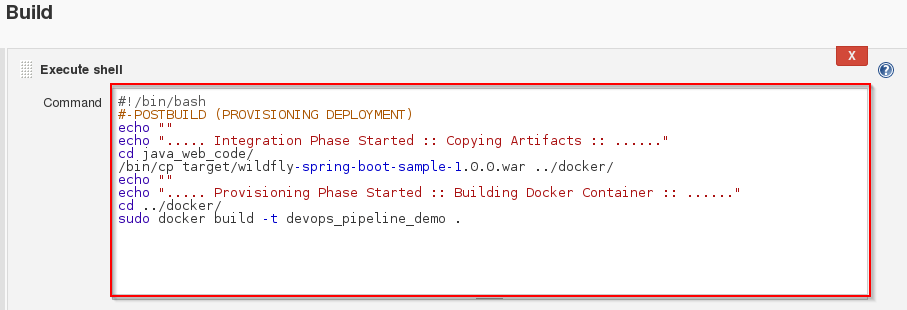
Step 9:**Select**Source Code Management**and provide the**Git**repository. Click on**Apply**and**Save**button.**



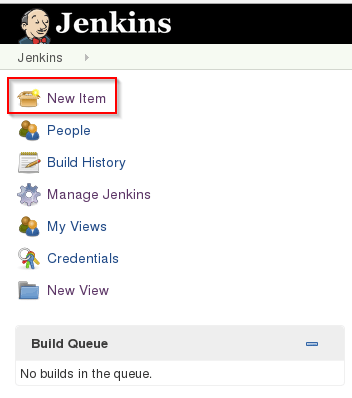
Step 10:**Then click on**Build->Select Execute Shell**.**



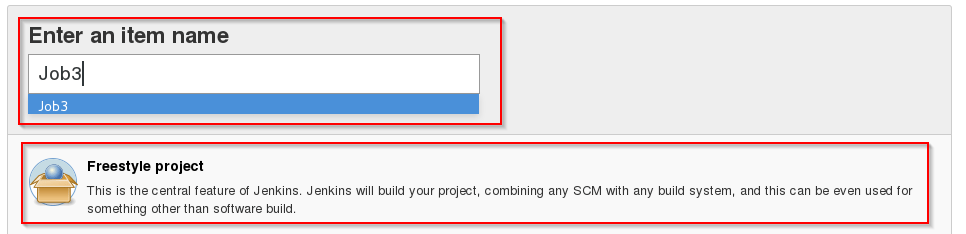
**Step 11:** Provide the shell commands. Here it will start the integration phase and **build** the [Docker Container](https://www.edureka.co/blog/what-is-docker-container?utm_source=medium&utm_medium=content-link&utm_campaign=ci-cd-pipeline" \t "_blank).



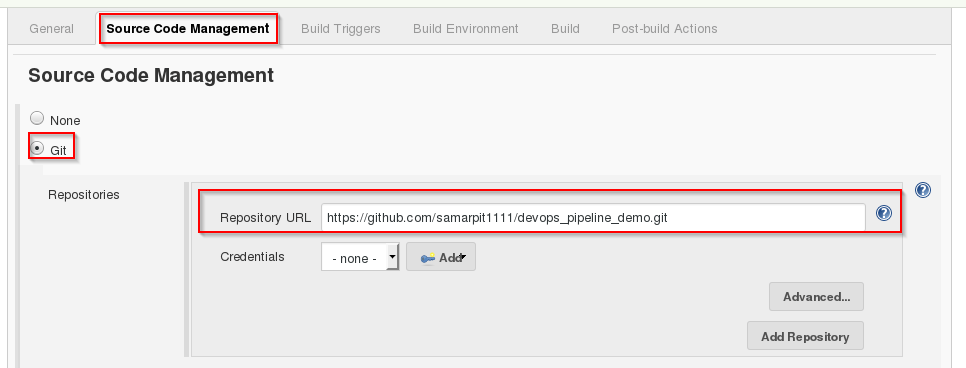
**Step 12:** Create the new **Job** by clicking on New Item.



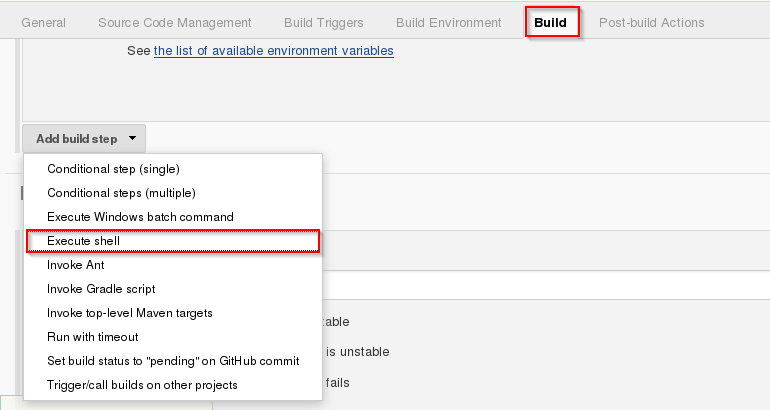
Step 13:**Select**freestyle**project and provide the item name (here I have given Job3) and click on OK.**



Step 14:**Select**Source Code Management**and provide the**Git**repository. Click on**Apply**and**Save**button.**



**Step 15:** Then click on **Build->Select Execute Shell**.



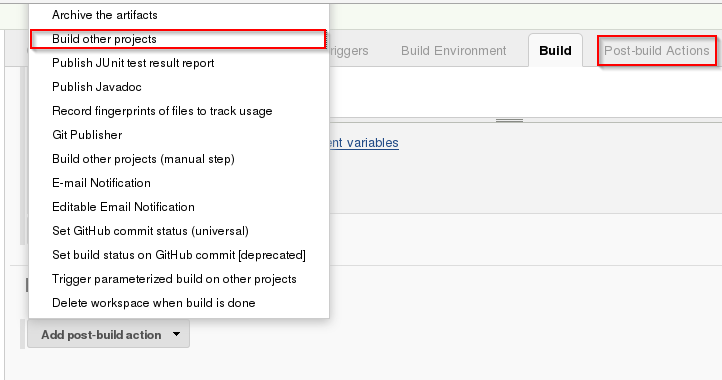
Step 16:**Provide the shell commands. Here it will check for the Docker Container file and then deploy it on port number 8180. Click on Save button.**



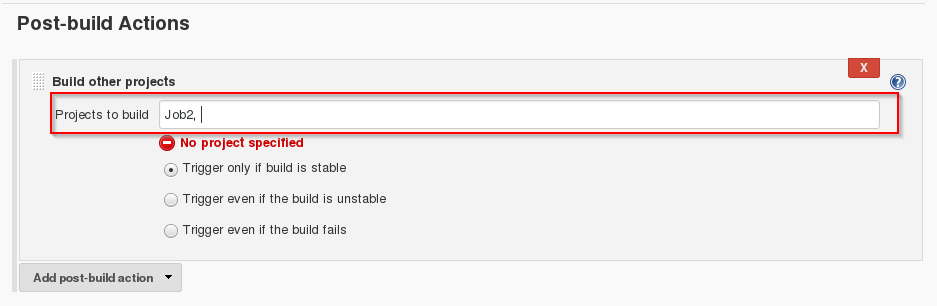
**Step 17:** Now click on **Job1 -> Configure**.



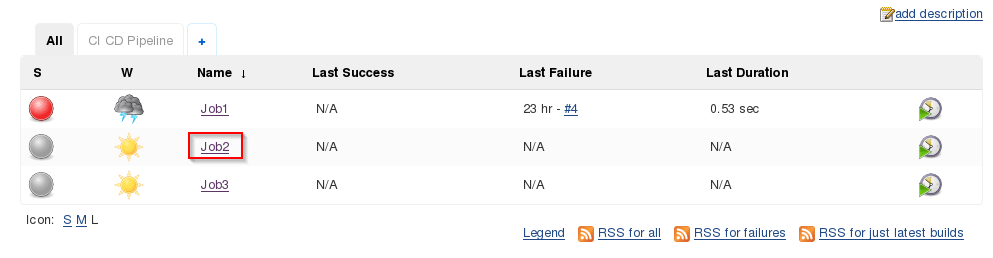
Step 18:**Click on**Post-build Actions -> Build other projects**.**

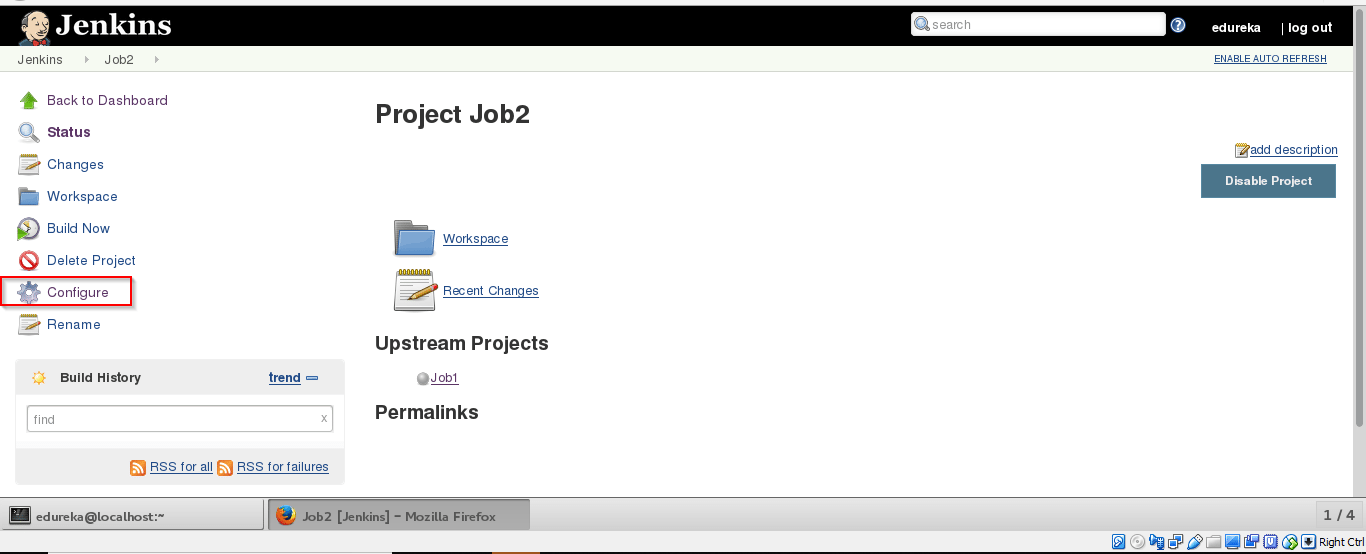


**Step 19:** Provide the project name to build after Job1 (here is Job2) and then click on **Save**.

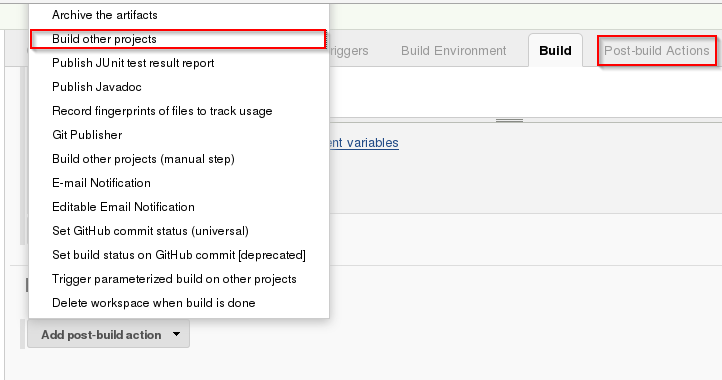


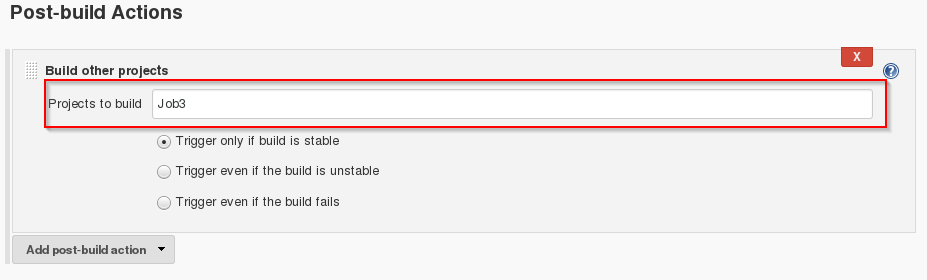
**Step 20:**Now click on **Job2 -> Configure**.



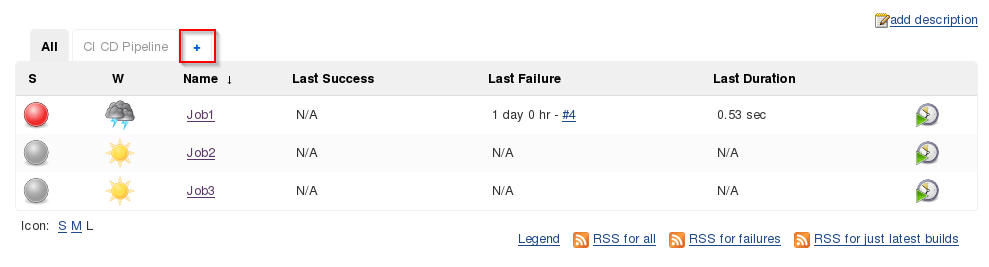


**Step 21:** Click on **Post-build Actions -> Build other projects**.

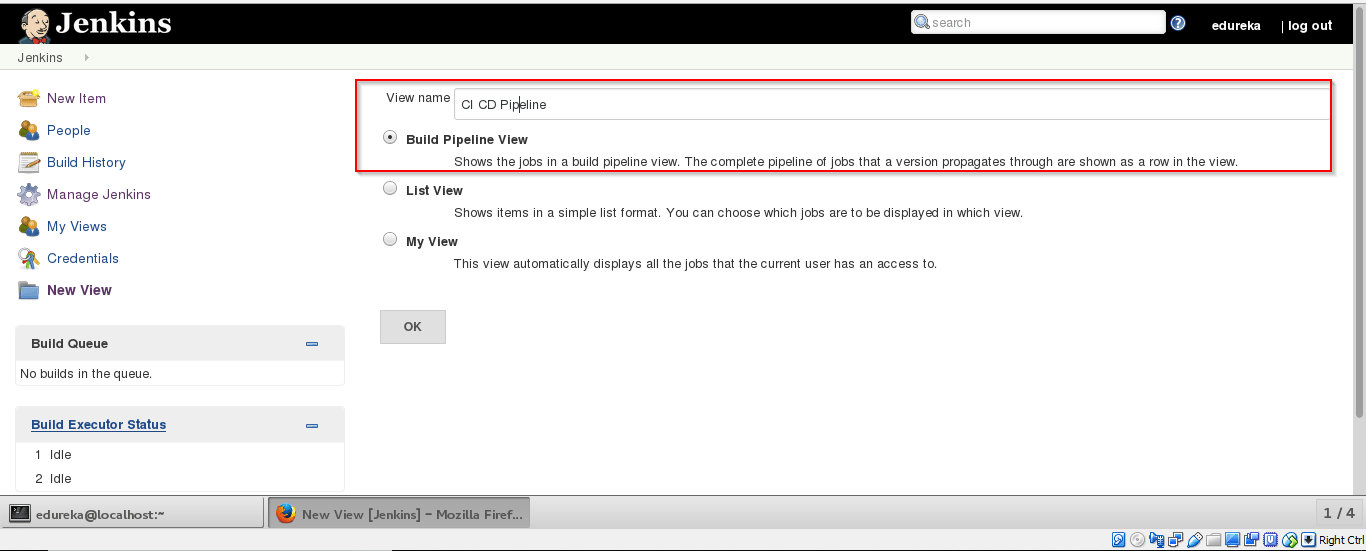


**Step 22:** Provide the project name to build after Job2 (here is Job3) and then click on **Save**.

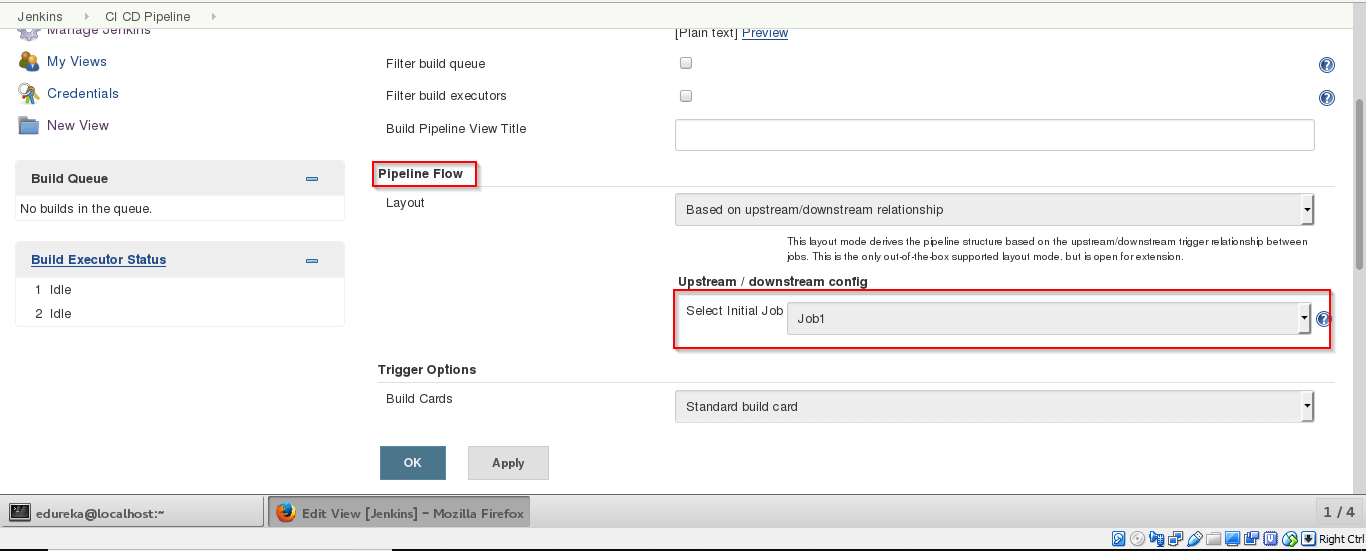
**Step 23:** Now we will be creating a Pipeline view. Click on ‘+’ sign.



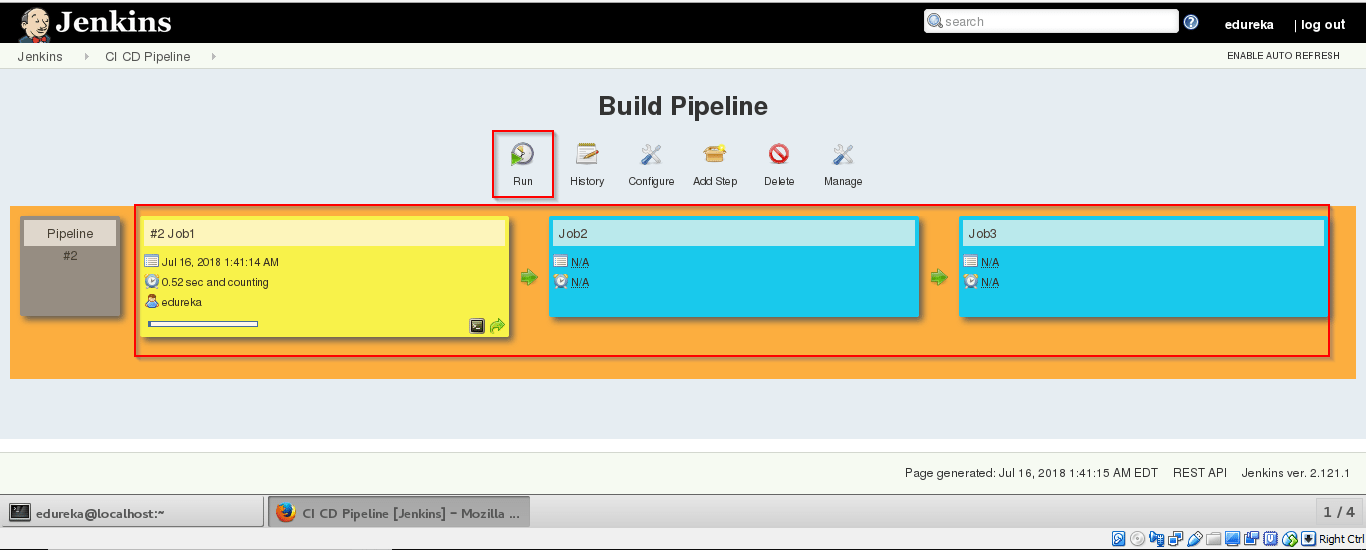
Step 24:**Select**Build Pipeline View**and provide the view name (here I have provided CI CD Pipeline).**



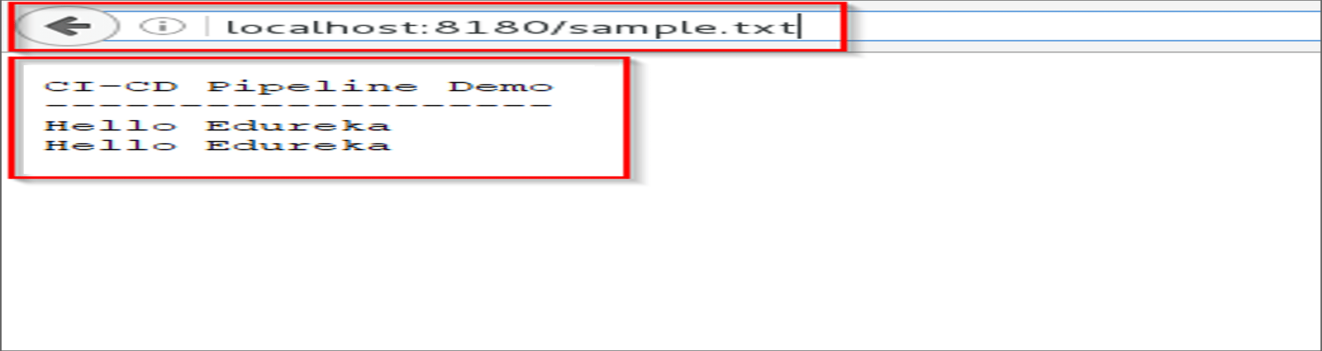
**Step 25:** Select the **initial** **Job** (here I have provided Job1) and click on OK.



Step 26:**Click on**Run **button to start the CI CD process.**



**Step 27:** After successful build open **localhost:8180/sample.text**. It will run the application.



So far, we have learned how to create CI CD Pipeline using Docker and Jenkins. The intention of DevOps is to create better-quality software more quickly and with more reliability while inviting greater communication and collaboration between teams.