Building a Continuous Delivery Pipeline Using Jenkins

(/users/3235277/saurabh-1.html) by Saurabh K (/users/3235277/saurabh-1.html) MVB

.

Aug. 09, 18 · DevOps Zone (/devops-tutorials-tools-news) · Tutorial

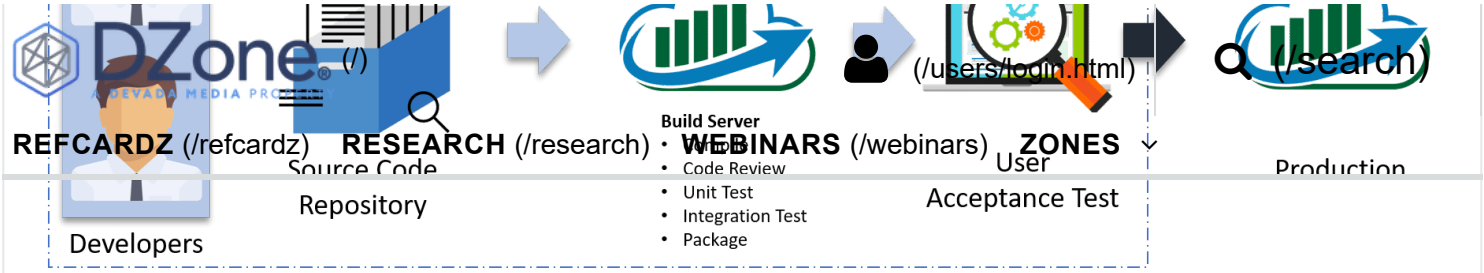
Continuous Delivery is a process, where code changes are automatically built, tested, and prepared for a release to production. I hope you have enjoyed my previous blogs on Jenkins. (https://www.edureka.co/blog/what-is-jenkins/) Here, I will talk about the following topics:

- 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.

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.

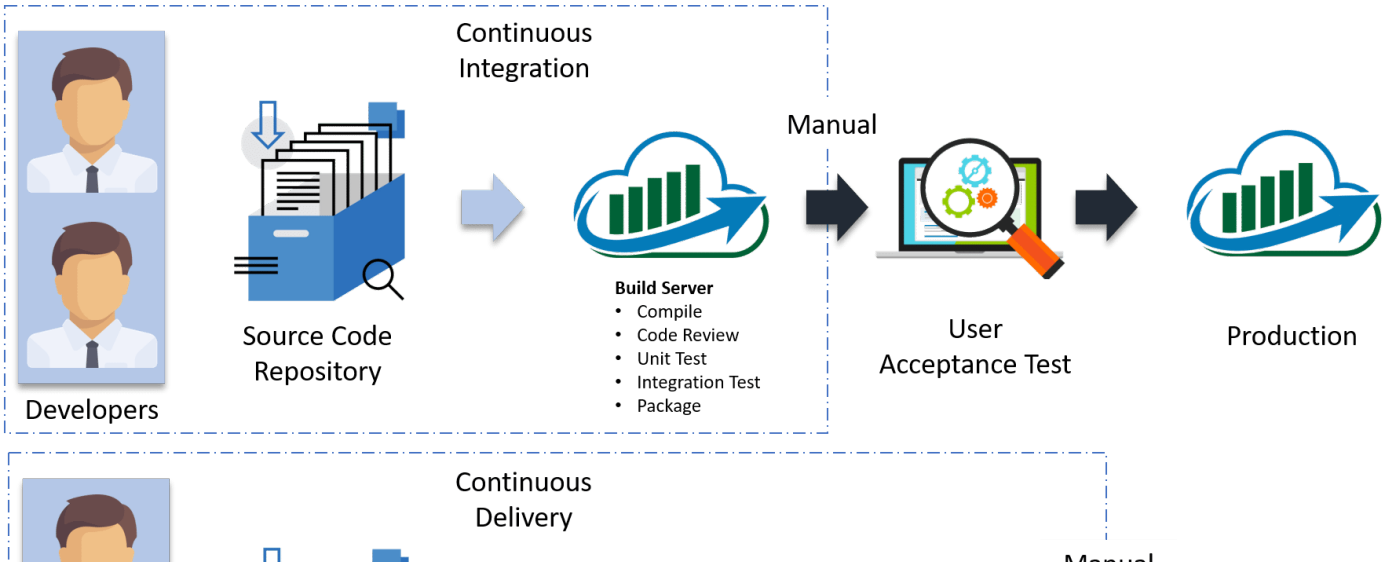
There are multiple tests that fall under this category. I will focus on a few, which are important for you to know, in order to understand this blog:

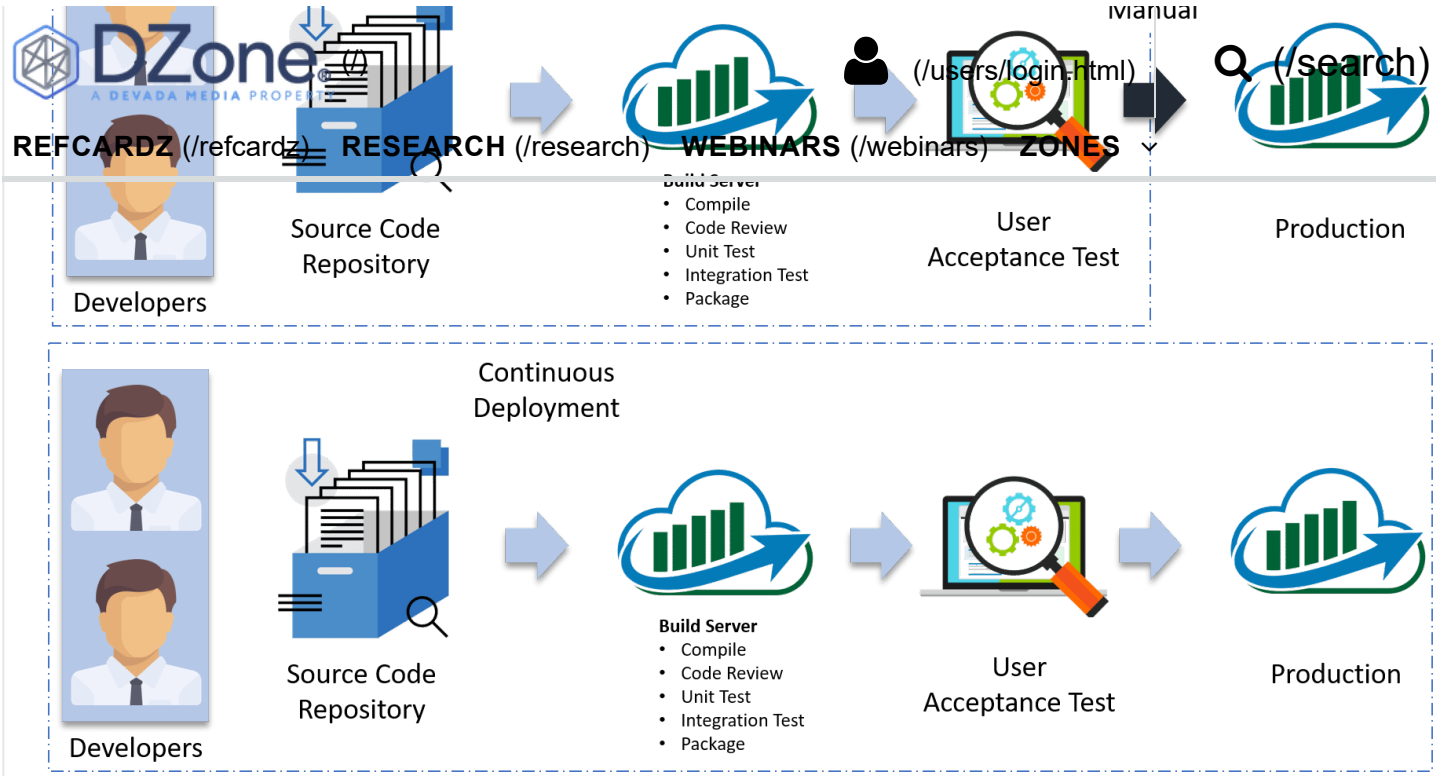
- **Functional/ Acceptance Testing:** It ensures that the specified functionality required in the system requirements works. It is done to make sure that the delivered product meets the requirements and works as the customer expected
- **System Testing:** It ensures that by putting the software in different 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 a 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 built 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:

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.

But the question is, whether Continuous Integration is enough.

Why We Need Continuous Delivery

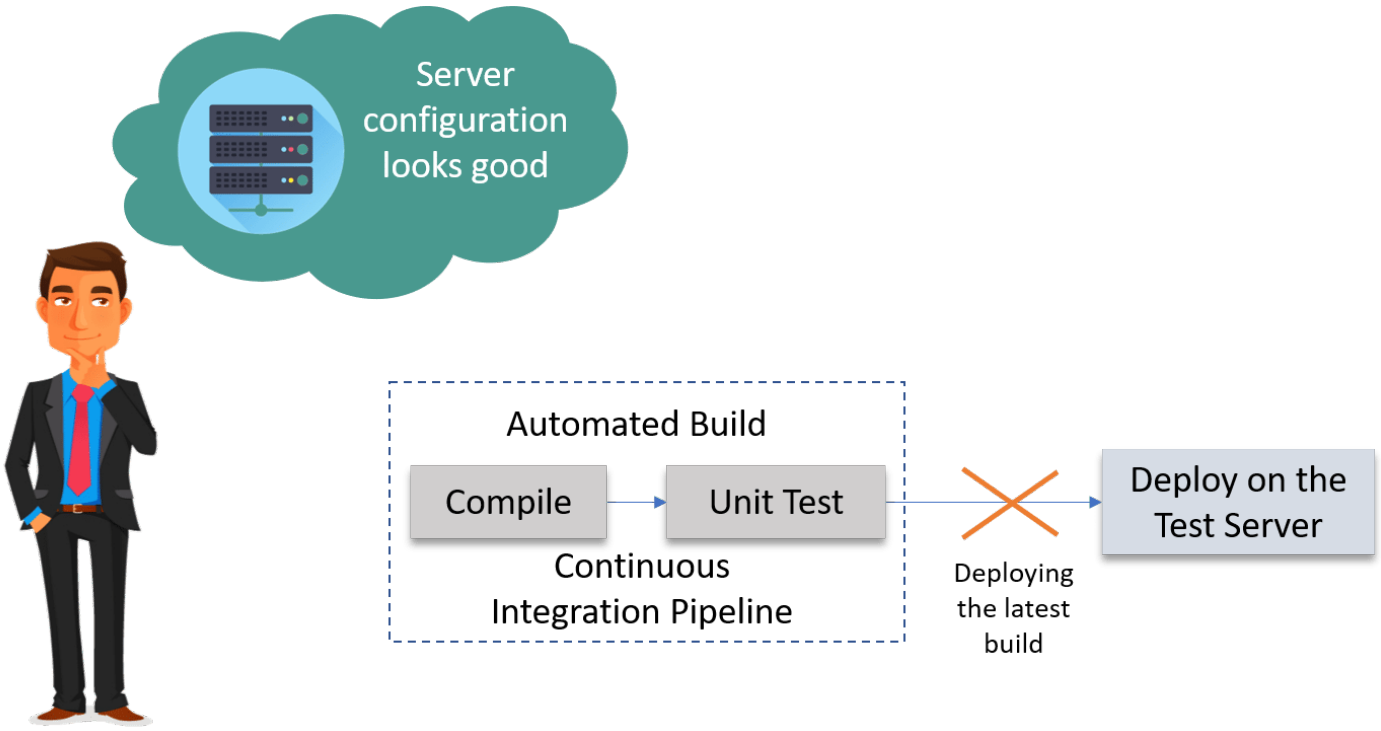
Let us understand this with an example. Imagine there are 80 developers working on a

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.



One Perceptive Developer Took a Smart Approach

Then one of the senior developers 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.

Without running Acceptance Tests in a production-like environment, they know

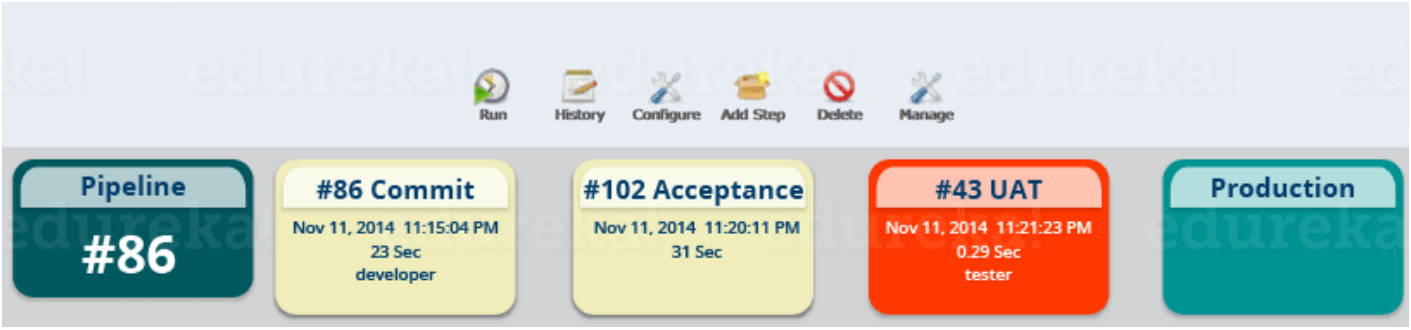
Without running Acceptance Tests in a production-like environment, they know nothing about whether the application meets the customer's specifications or whether it can be deployed and survive in the real world. If they want timely feedback on these topics, they must extend the range of their continuous 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 to 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:

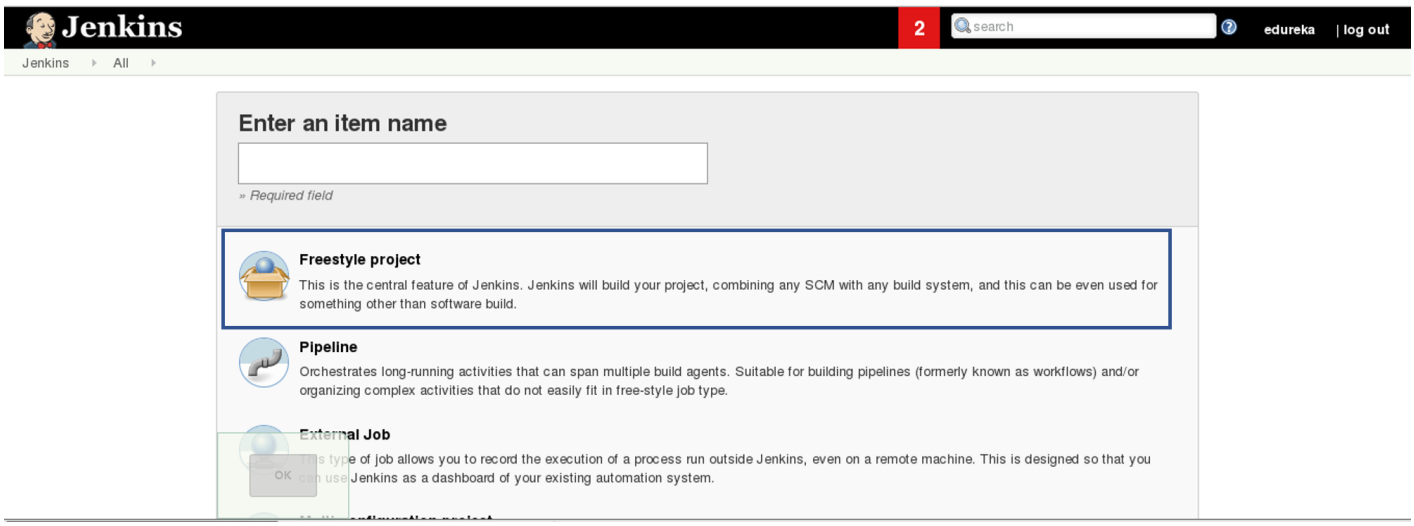
Steps Involved in the Demo

- 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 serve

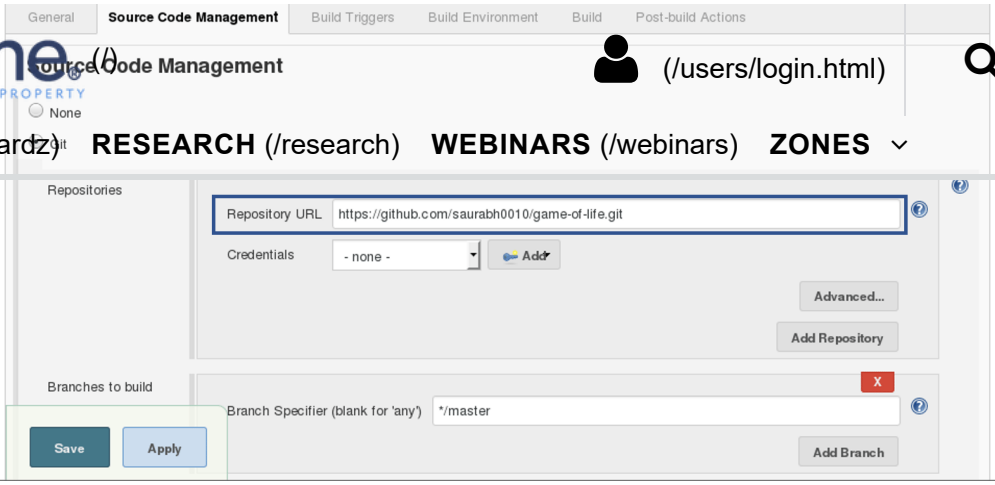
Step 1 — Compiling the Source Code

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

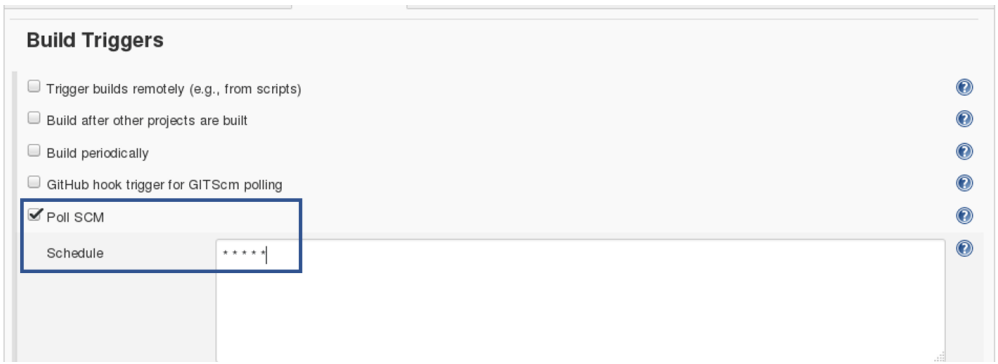
Give a name to your project and select Freestyle Project:



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:

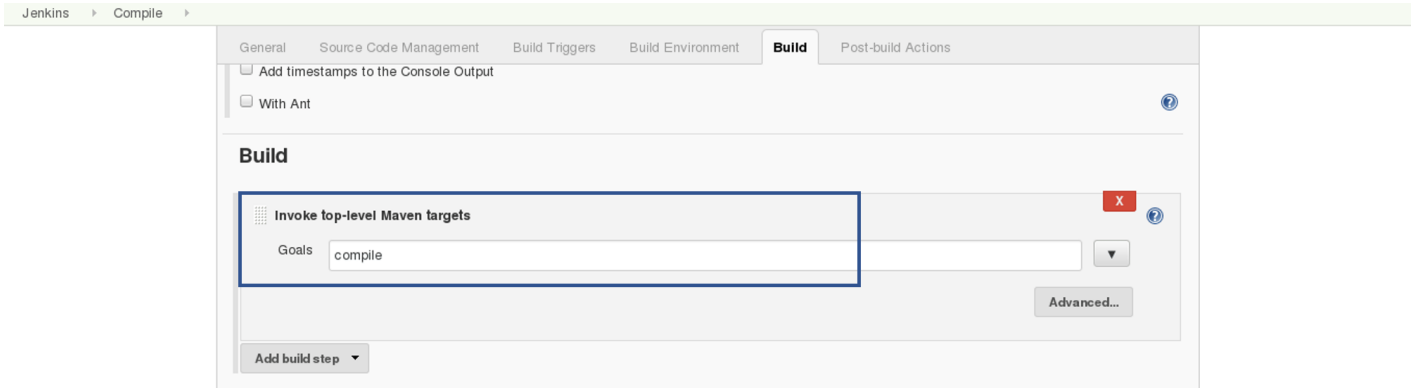
```
1 mvn clean package
```

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:

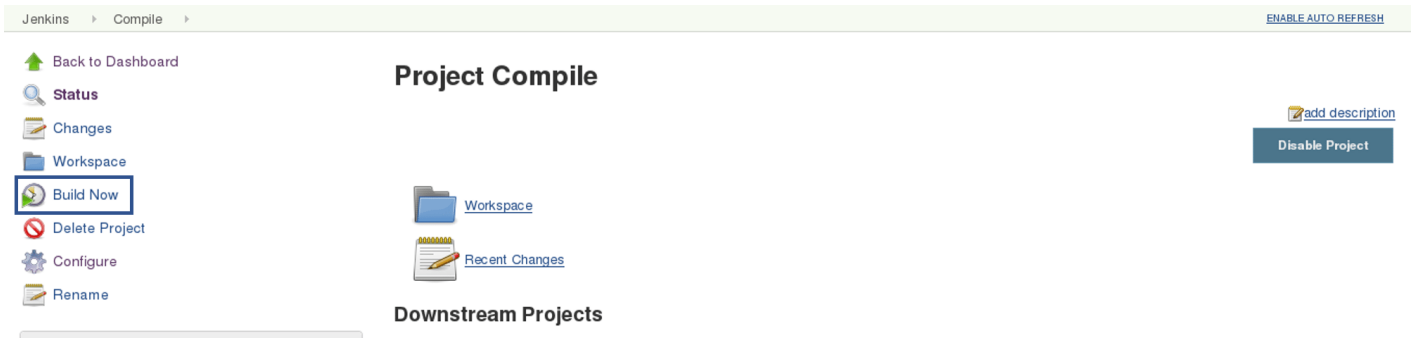
```
1 compile
```

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.

```
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ gameoflife-web ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /var/lib/jenkins/workspace/Compile/gameoflife-web/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] gameoflife ......................................... SUCCESS [1.140s]
[INFO] gameoflife-build ................................... SUCCESS [0.398s]
[INFO] gameoflife-core .................................... SUCCESS [1.112s]
[INFO] gameoflife-web ..................................... SUCCESS [0.320s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.570s
[INFO] Finished at: Fri Jul 27 02:54:42 EDT 2018
[INFO] Final Memory: 17M/204M
[INFO] ------------------------------------------------------------------------
Finished: SUCCESS
```

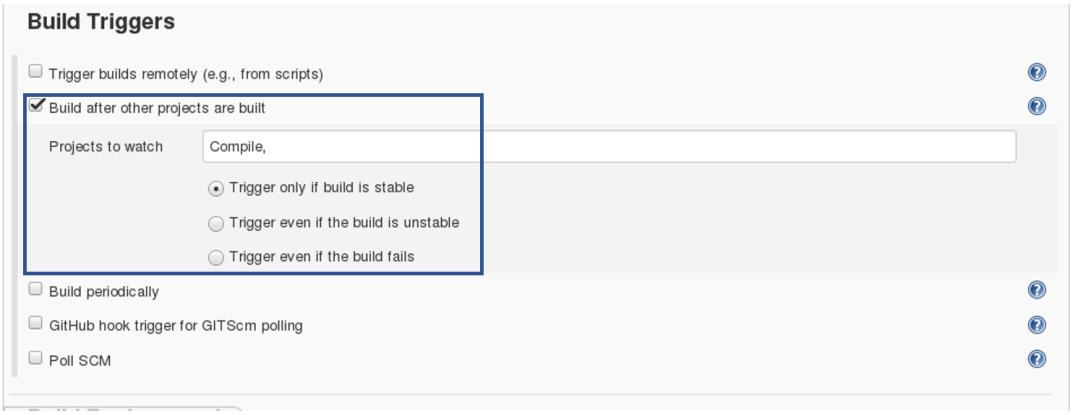
Now we will create one more Freestyle Project for unit testing.

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

Now, in the "Build 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:

```
1 test
```

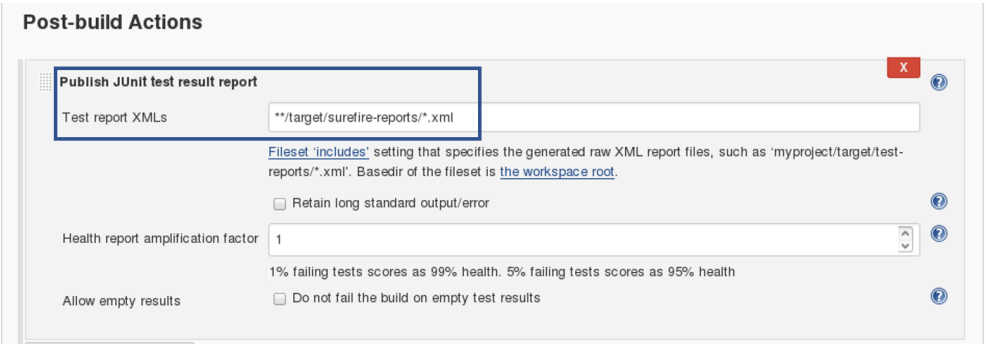
Jenkins also does a great job of helping you display your test results and test result trends.

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. 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.

```
1 **/target/surefire-reports/*.xml
```



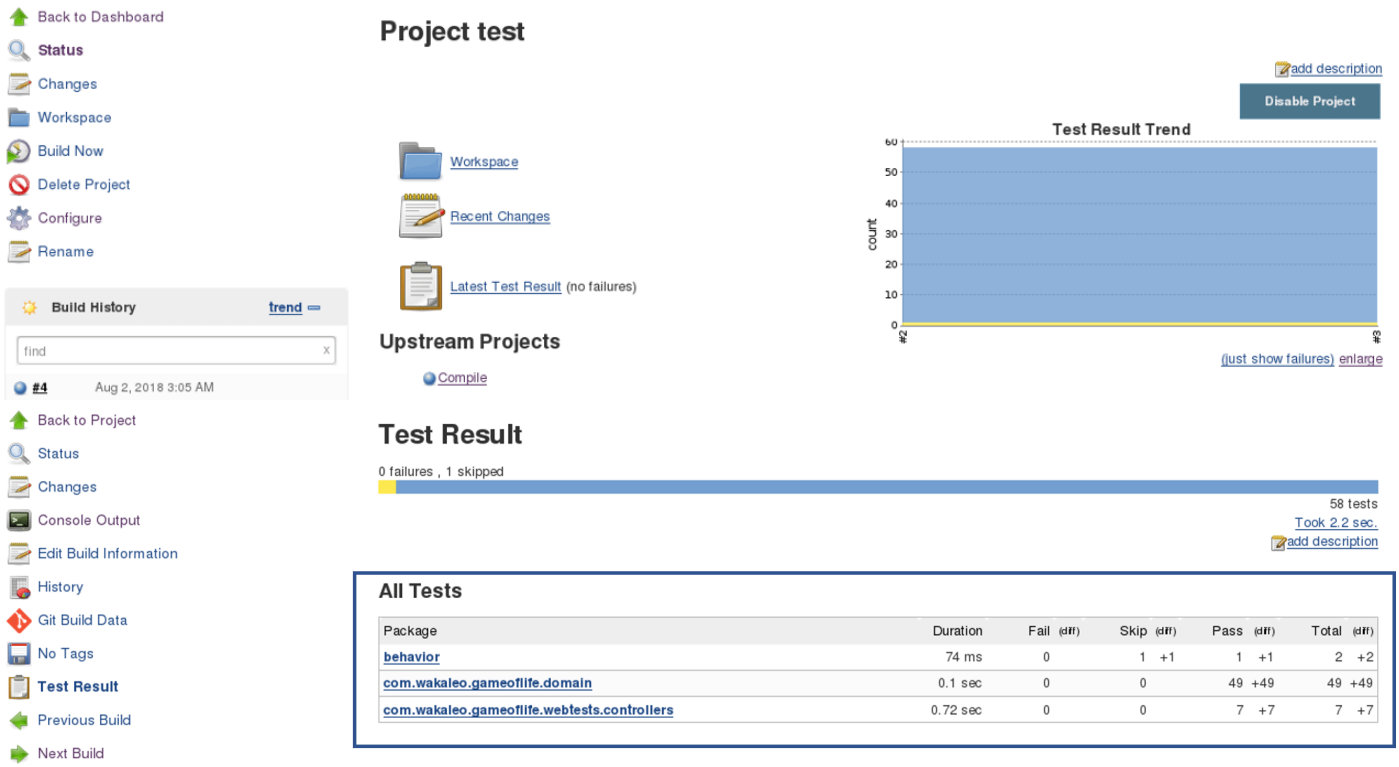
Again, save it and click on Build Now.

```
[INFO]
[INFO] --- maven-easyb-plugin:1.4:test (default) @ gameoflife-web ---
[INFO] /var/lib/jenkins/workspace/test/gameoflife-web/src/test/stories does not exists.  Skipping easyb testing
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] gameoflife ......................................... SUCCESS [5.834s]
[INFO] gameoflife-build ................................... SUCCESS [2.045s]
[INFO] gameoflife-core .................................... SUCCESS [8.571s]
[INFO] gameoflife-web ..................................... SUCCESS [2.556s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 20.467s
[INFO] Finished at: Thu Aug 02 03:05:59 EDT 2018
```

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

```
[edureka@localhost surefire-reports]$ ls
com.wakaleo.gameoflife.domain.WhenYouCreateACell.txt
com.wakaleo.gameoflife.domain.WhenYouCreateAGrid.txt
com.wakaleo.gameoflife.domain.WhenYouCreateANewUniverse.txt
com.wakaleo.gameoflife.domain.WhenYouPlayTheGameOfLife.txt
com.wakaleo.gameoflife.domain.WhenYouPrintAGrid.txt
com.wakaleo.gameoflife.domain.WhenYouReadAGridFromAString.txt
TEST-behavior.CountingThings.xml
TEST-behavior.MultiplyingThings.xml
TEST-com.wakaleo.gameoflife.domain.WhenYouCreateACell.xml
TEST-com.wakaleo.gameoflife.domain.WhenYouCreateAGrid.xml
TEST-com.wakaleo.gameoflife.domain.WhenYouCreateANewUniverse.xml
TEST-com.wakaleo.gameoflife.domain.WhenYouPlayTheGameOfLife.xml
TEST-com.wakaleo.gameoflife.domain.WhenYouPrintAGrid.xml
TEST-com.wakaleo.gameoflife.domain.WhenYouReadAGridFromAString.xml
```

In the Jenkins dashboard, you will also notice the test results:

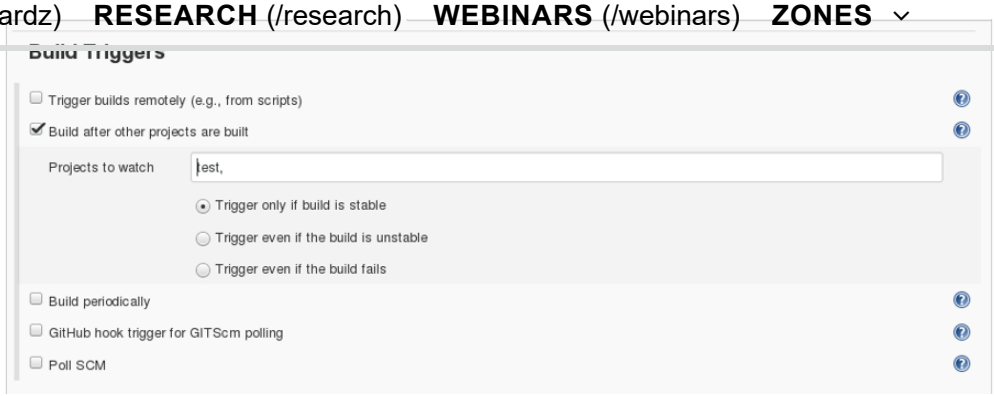


Step 3 — Creating a WAR File and Deploying on the Tomcat Server

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:

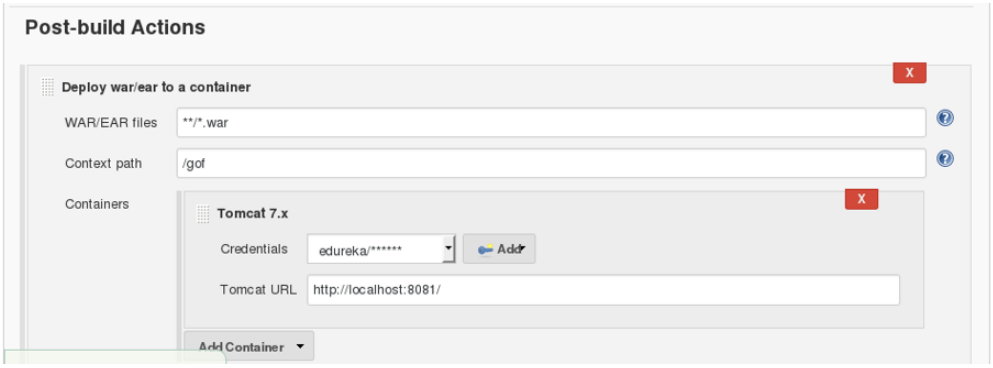


Basically, after the test job, the deployment phase will start automatically.

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

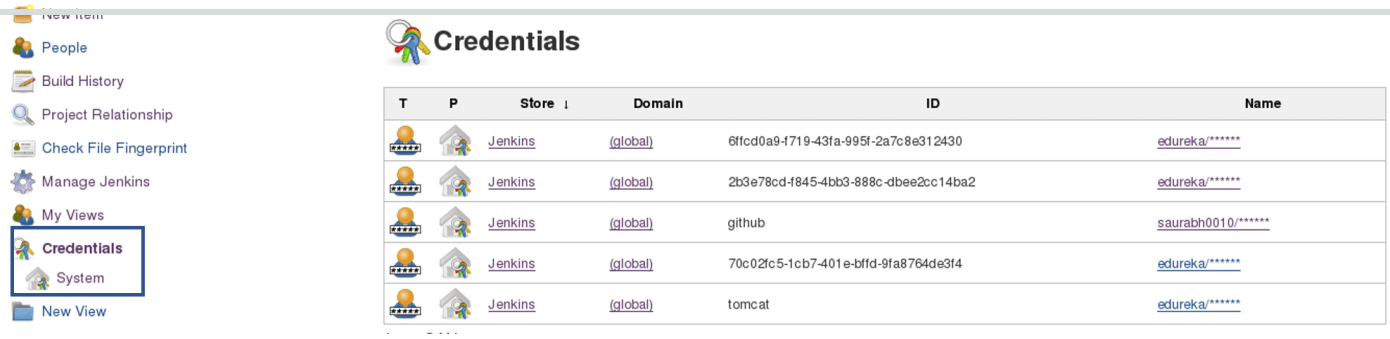
```
1 mvn package
```

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:

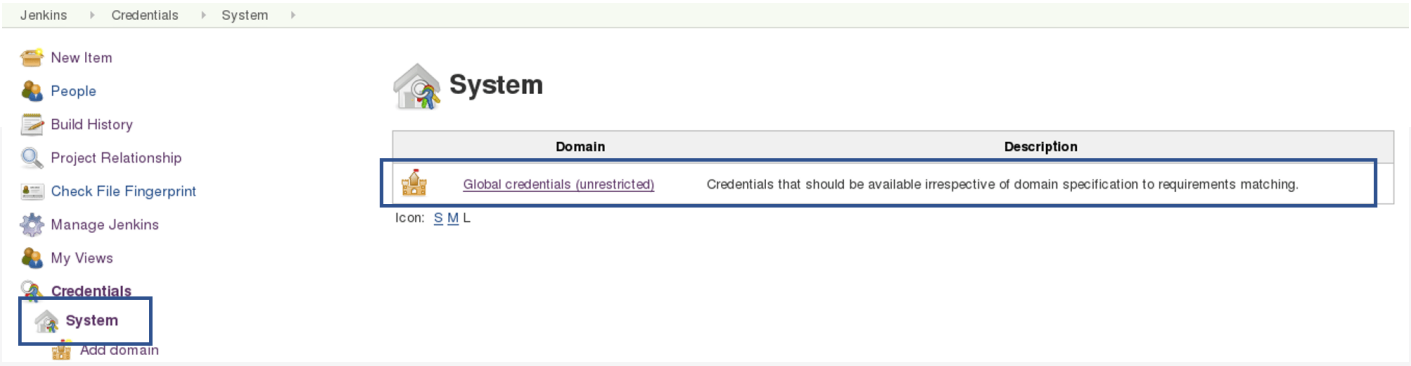


Select the Tomcat credentials and, notice the above screenshot. Also, you need to give the URL of your Tomcat server.

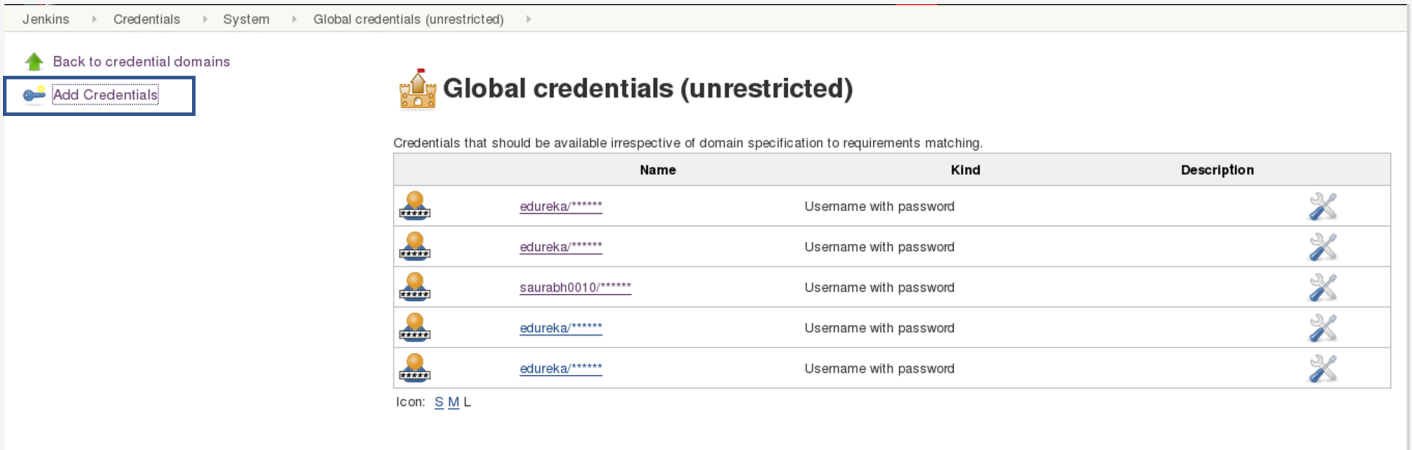
In order to add credentials in Jenkins, click on credentials option on the Jenkins dashboard.



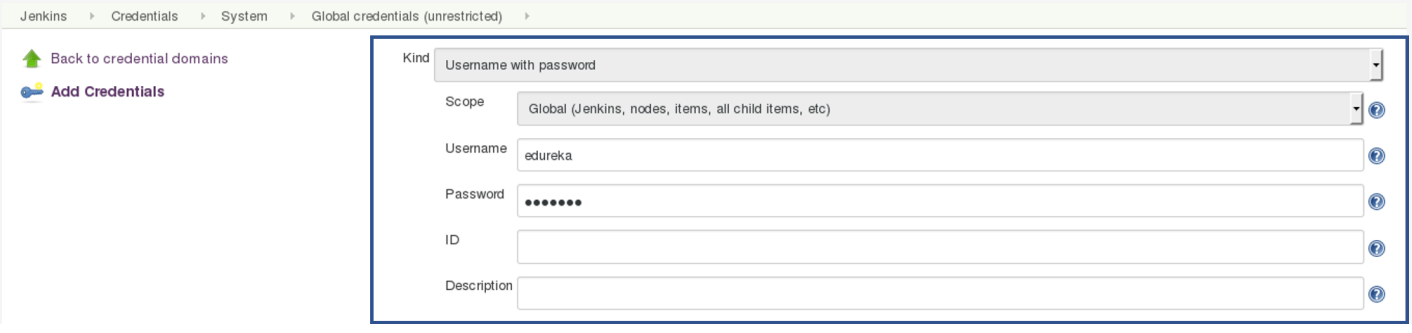
Click on System and select global credentials.



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

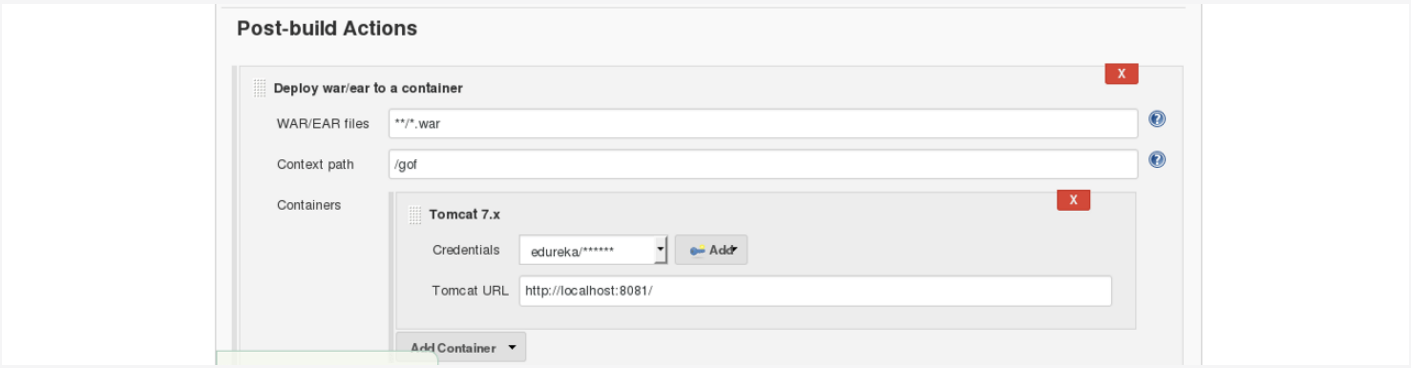


Add the Tomcat credentials, consider the below screenshot.

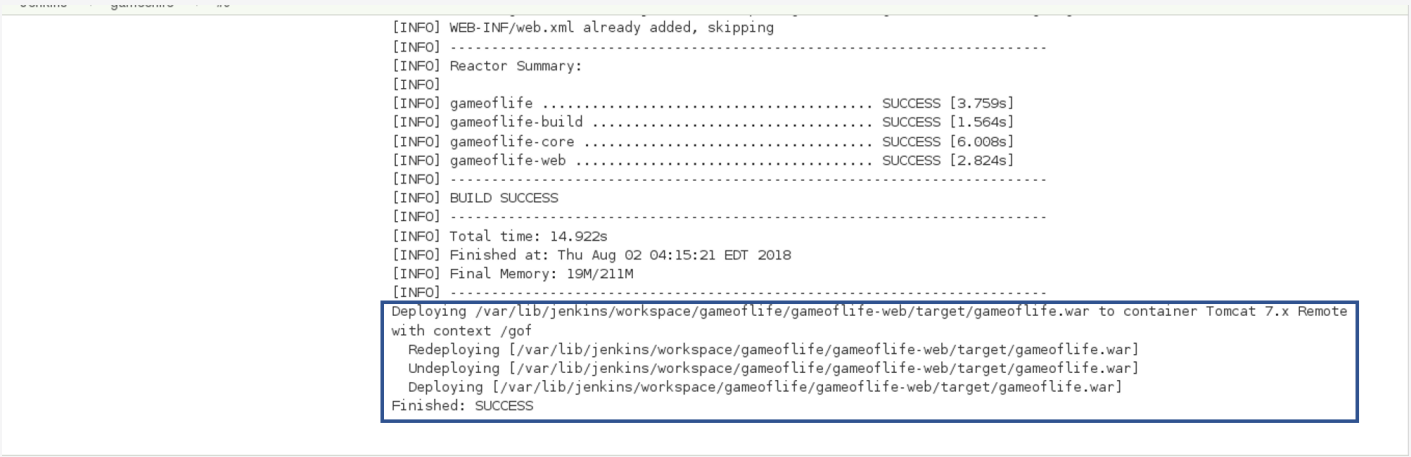


Click on OK.

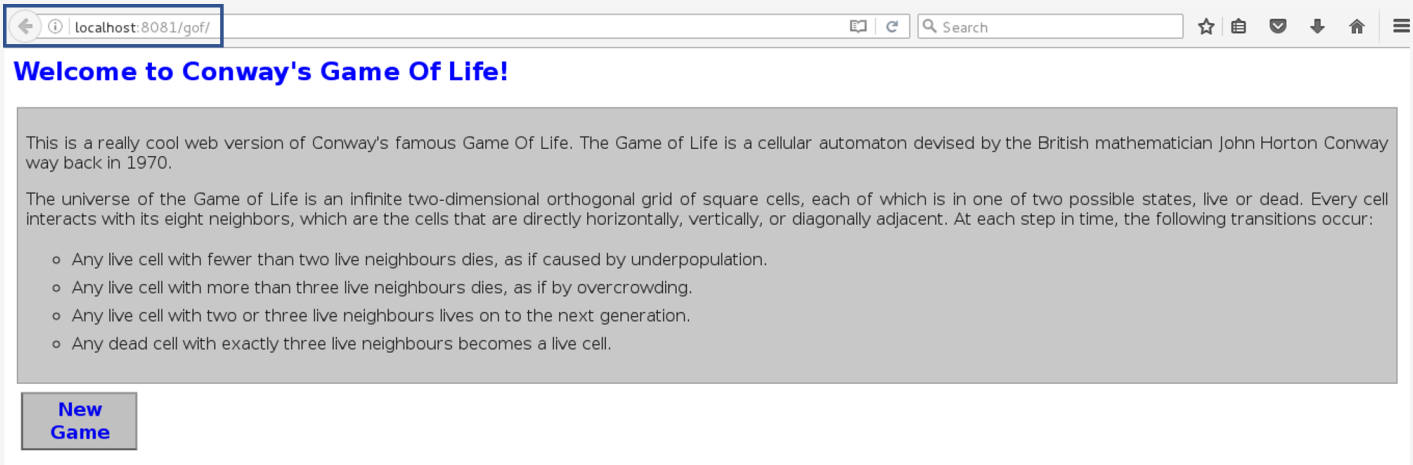
Now, in your Project Configuration, add the tomcat credentials which you have inserted in the previous step.



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:

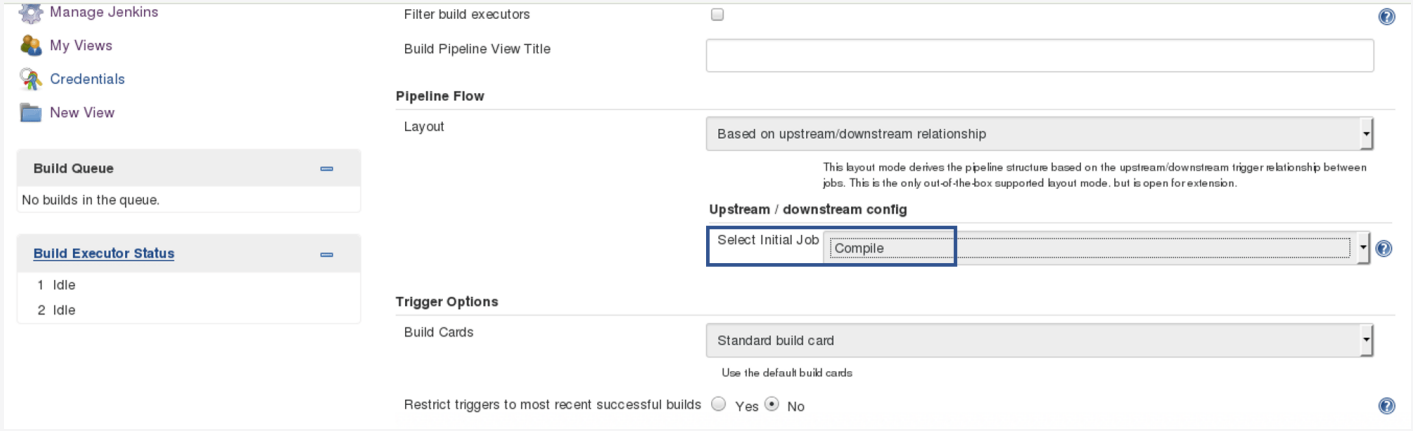


I hope you have understood the meaning of the context path.

Now, to create a pipeline view, consider the below screenshot:

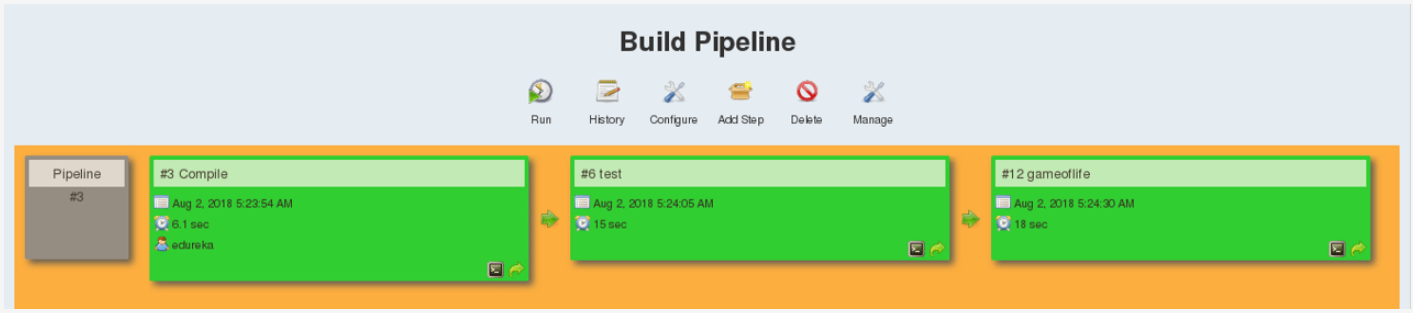
Click on the plus icon to create a new view.

Configure the pipeline the way you want:



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.



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

Topics: DEVOPS, CONTINUOUS DELIVERY, CONTINUOUS DEPLOYMENT, JENKINS, PIPELINES, TUTORIAL, CI/CD