



DZone (/) > DevOps Zone (/devops-tutorials-tools-news) > Building a Continuous Delivery Pipeline Using Jenkins

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

Like (18) Comment (7) Save Tweet

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:

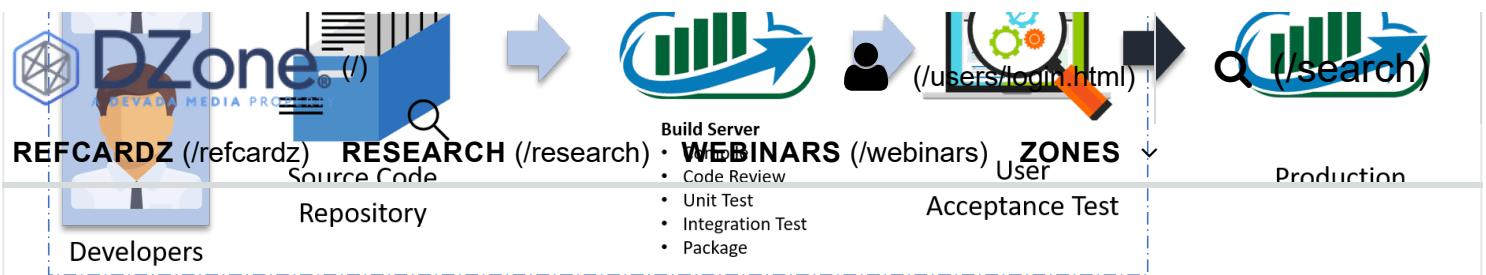
- What is Continuous Delivery?
- Types of Software Testing
- Difference Between Continuous Integration, Delivery, and Deployment
- What is the need for Continuous Delivery?
- Hands-on Using Jenkins and Tomcat

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

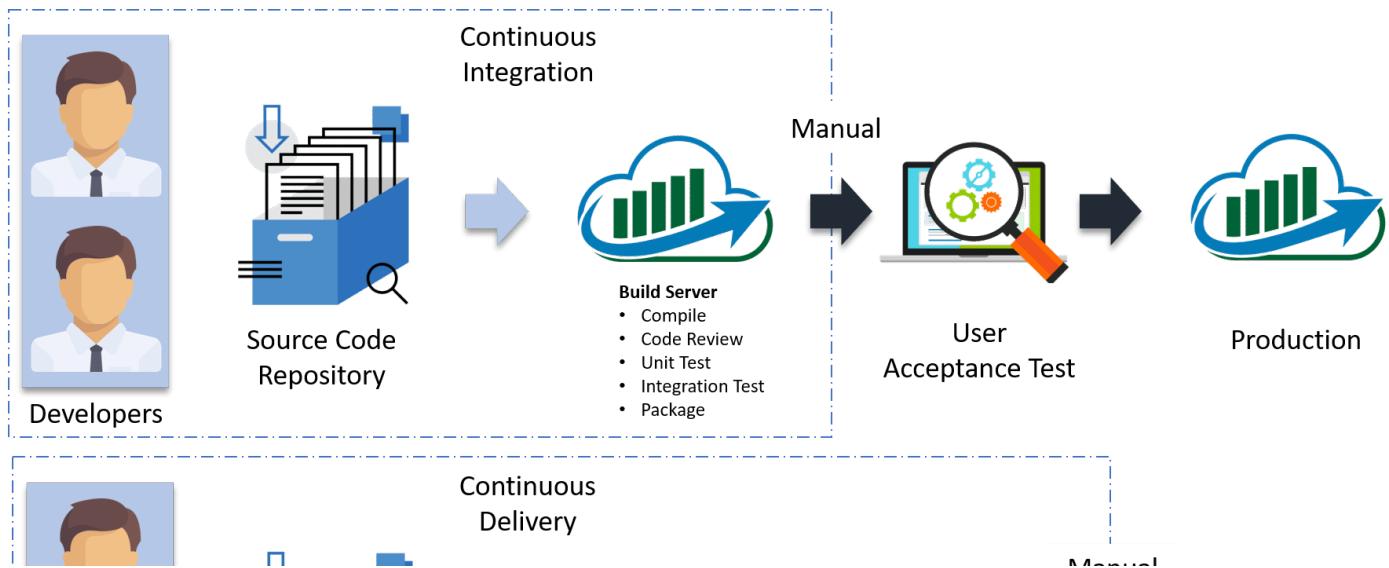
There are multiple tests that fall under this category. I will focus on a few, which are important for you to know, in order to understand this blog:

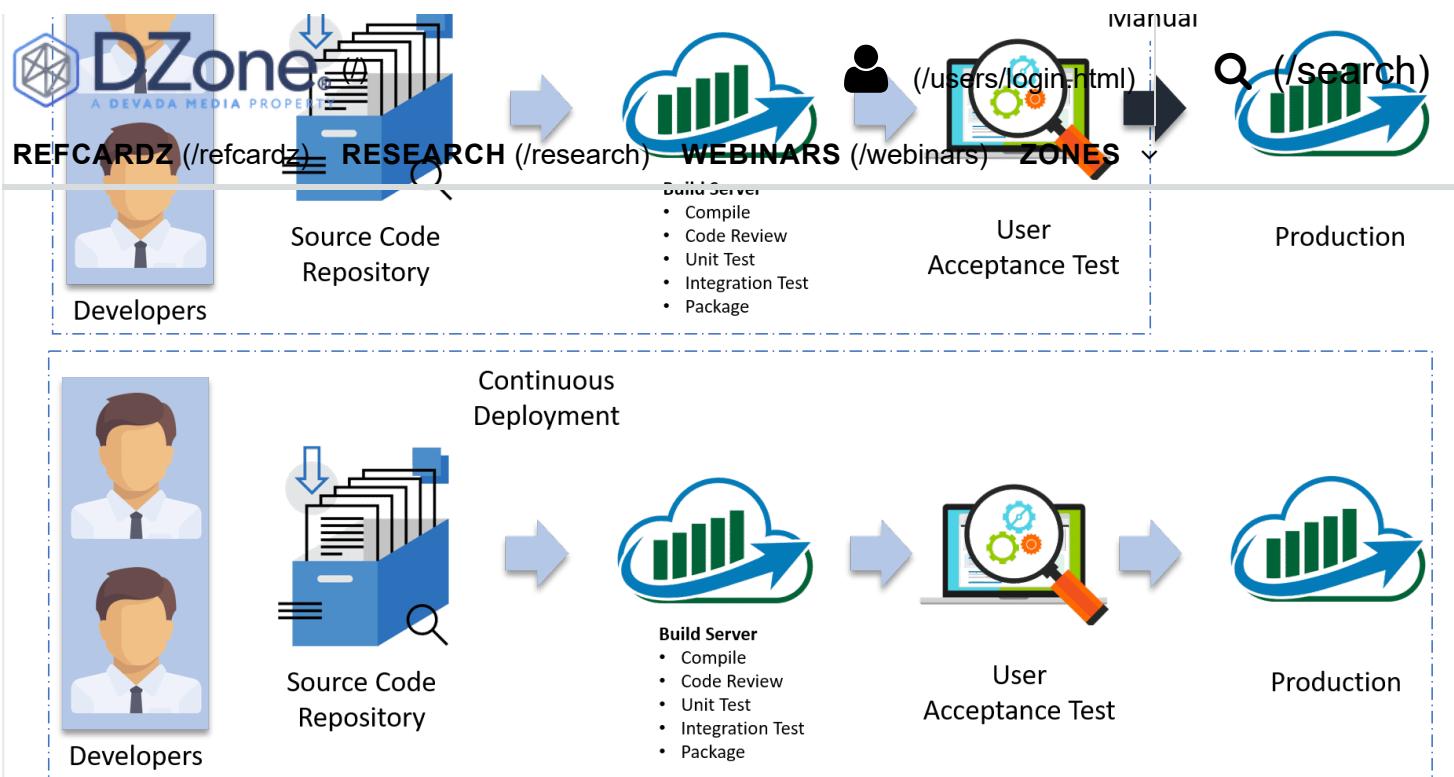
- **Functional/ Acceptance Testing:** It ensures that the specified functionality required in the system requirements works. It is done to make sure that the delivered product meets the requirements and works as the customer expected
- **System Testing:** It ensures that by putting the software in different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

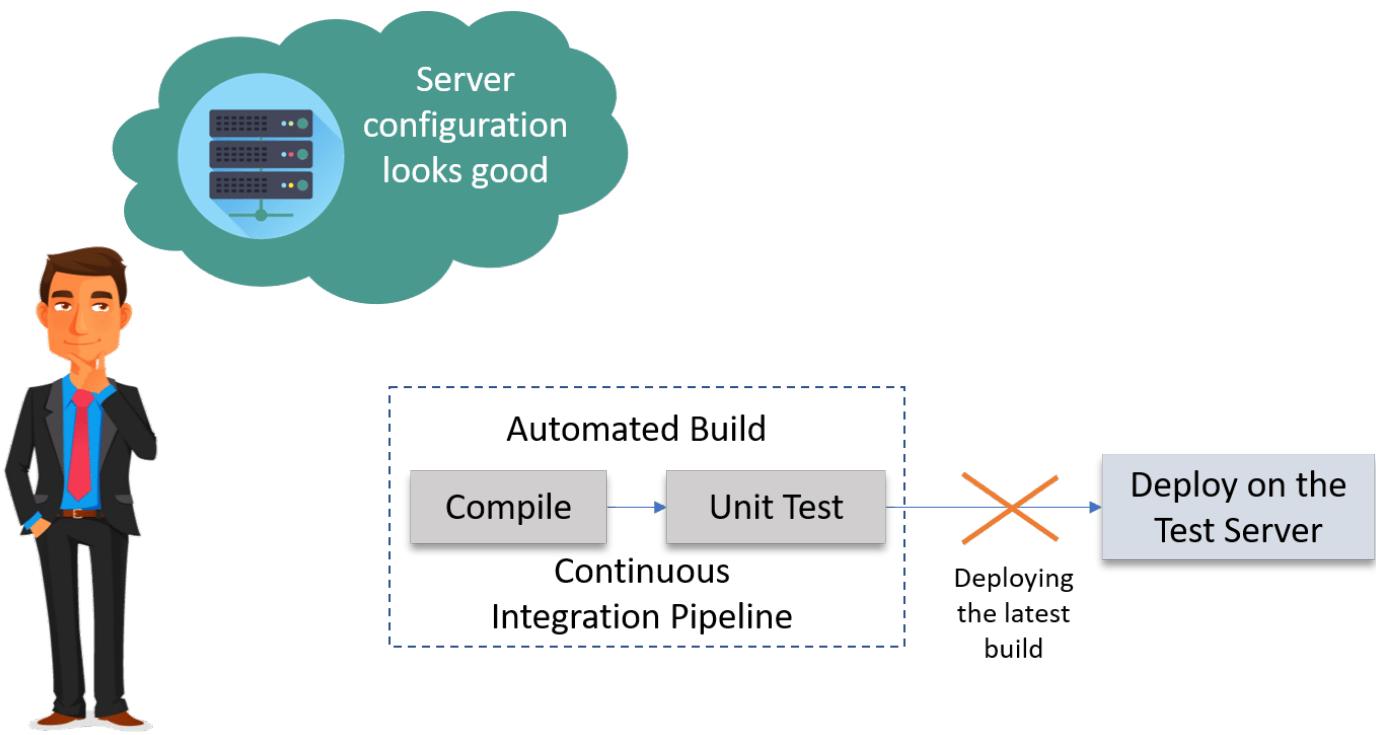
large project. They are using Continuous Integration pipelines in order to facilitate automated builds. We know build includes Unit Tests and in one day they decided to deploy the latest build that had passed the unit tests into a test environment.

REFCARDZ (/refcardz) RESEARCH (/research) WEBINARS (/webinars) ZONES ▾

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.

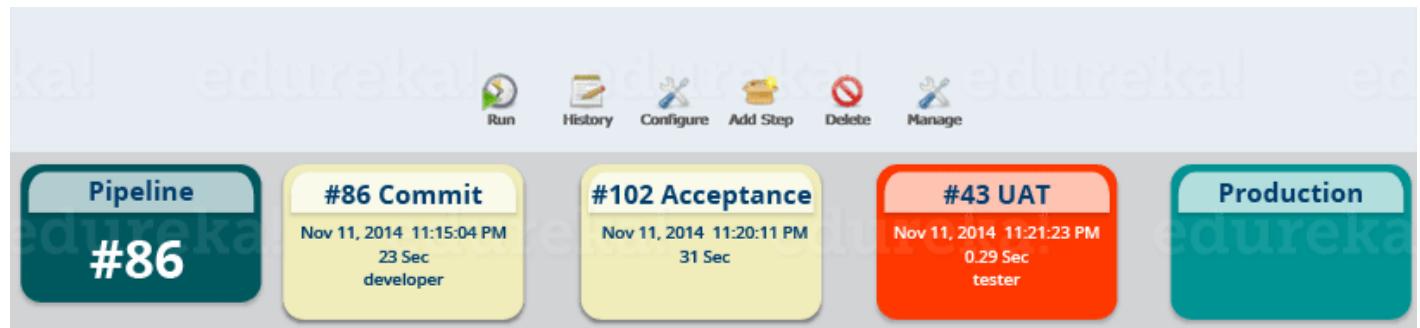
He stepped back through earlier and earlier versions until he found that the system had stopped working three weeks earlier. A tiny, obscure bug had prevented the system from starting correctly. Although, the project had good unit test coverage. Despite this, 80 developers, who usually only ran the tests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

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 we will be using Jenkins to create a Continuous delivery Pipeline, which will include the following tasks:

REFCARDZ (/refcardz) RESEARCH (/research) WEBINARS (/webinars) ZONES ▾

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



Step 1 — Compiling the Source Code

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

The screenshot shows the Jenkins dashboard with a black header. In the top left, there is a logo and the word "Jenkins". On the right side of the header, there are links for "edureka" and "log out". Below the header, there is a search bar with a magnifying glass icon and a link "ENABLE AUTO REFRESH". Underneath the header, there are two buttons: "New Item" (highlighted with a blue border) and "People". At the bottom of the dashboard, there are two tabs: "All" and a plus sign icon.

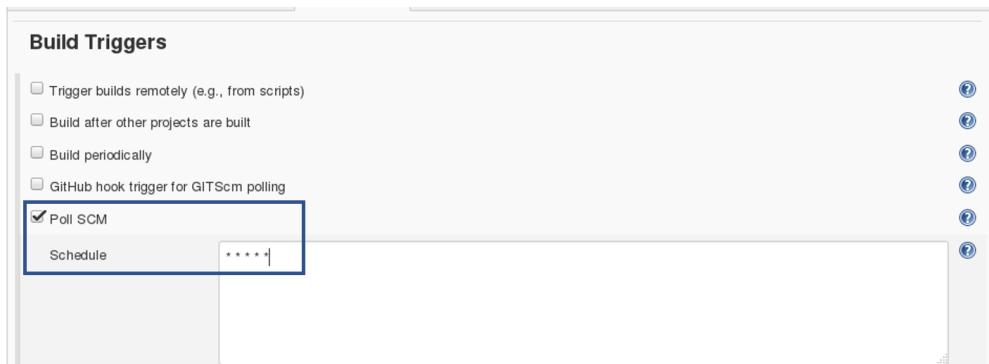
Give a name to your project and select Freestyle Project:

The screenshot shows the "Enter an item name" input field, which is empty. Below it, there is a note "» Required field". There are three options listed: "Freestyle project" (selected and highlighted with a blue border), "Pipeline" (described as orchestrating long-running activities), and "External Job" (described as allowing execution of processes outside Jenkins). The "Freestyle project" section includes a description: "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."

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:

The screenshot shows the 'Source Code Management' tab in the DZone interface. A GitHub repository URL is entered as `https://github.com/saurabh0010/game-of-life.git`. The 'Branches to build' section specifies the branch specifier as `*/master`. Buttons for 'Save' and 'Apply' are visible at the bottom.

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



(/users/login.html)



(/search)

- deploy — done in the build environment, copies the final package to the remote repository for sharing with other developers and projects

REFCARDZ ([refcardz](#)) RESEARCH ([research](#)) WEBINARS ([webinars](#)) ZONES [▼](#)

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:

The screenshot shows the Jenkins interface for a project named 'Compile'. The 'Build' tab is selected. Under the 'Build' section, there is a step titled 'Invoke top-level Maven targets' with the goal 'compile' specified. Other options like 'Add timestamps to the Console Output' and 'With Ant' are also visible.

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.

The screenshot shows the Jenkins 'Project Compile' page. On the left, there is a sidebar with links: Back to Dashboard, Status, Changes, Workspace, Build Now (which is highlighted), Delete Project, Configure, and Rename. The main area has sections for 'Workspace' and 'Recent Changes'. There are also links for 'Downstream Projects', 'add description', and 'Disable Project'.

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



(/users/login.html)

Q (/search)

DZone[®]
A NEVADA MEDIA PROPERTY
REFCARDZ (/refcardz) RESEARCH (/research) WEBINARS (/webinars) ZONES ▾
[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

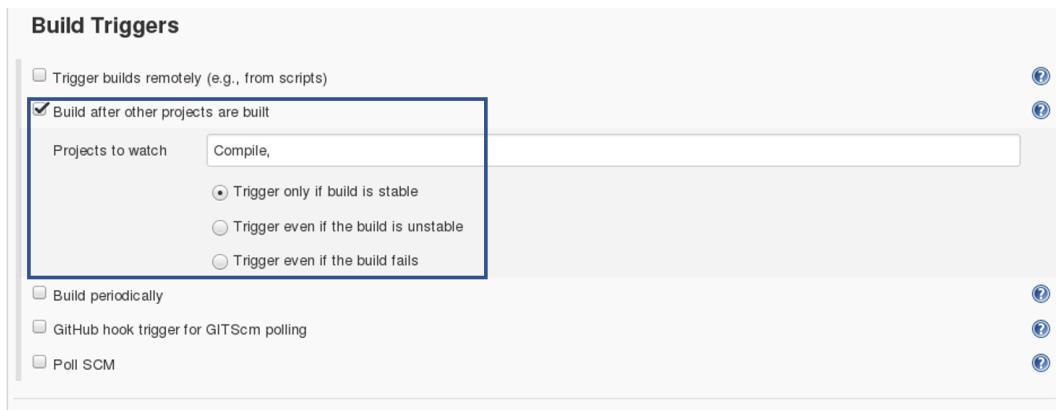
Now we will create one more Freestyle Project for unit testing.

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:

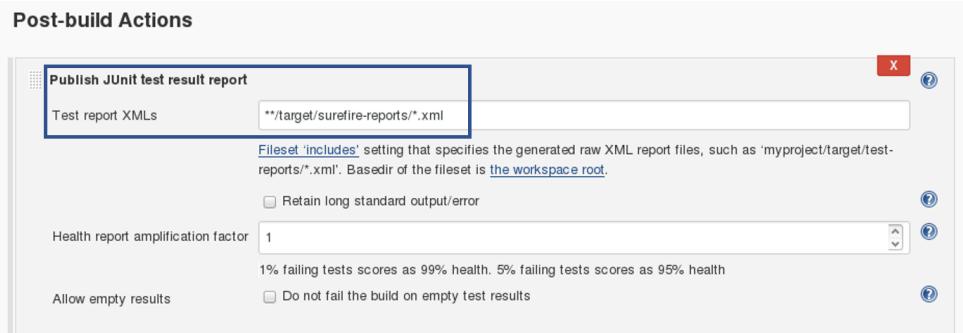
1 test

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.

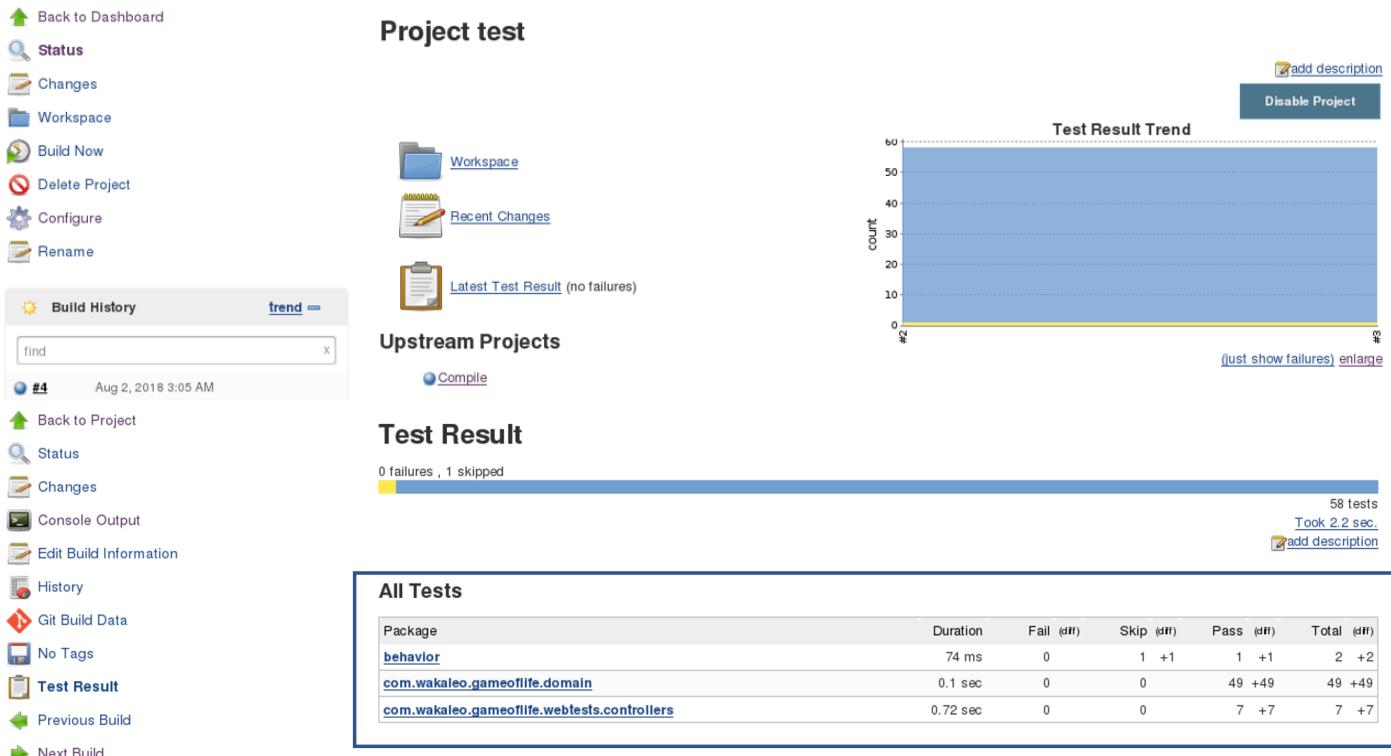
```
[INFO]
[INFO] --- maven-easyb-plugin:1.4:test (default) @ gameoflife-web ---
[INFO] /var/lib/jenkins/workspace/test/gameoflife-web/src/test/stories does not exists. Skipping easyb testing
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] gameoflife ..... SUCCESS [5.834s]
[INFO] gameoflife-build ..... 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
```



[REFCARDZ](#) (/refcardz) [RESEARCH](#) (/research) [WEBINARS](#) (/webinars) [ZONES](#) (/zones)
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:



The screenshot shows the Jenkins dashboard for a project named "Project test".

- Left Sidebar:**
 - Back to Dashboard
 - Status
 - Changes
 - Workspace
 - Build Now
 - Delete Project
 - Configure
 - Rename
- Build History:** Shows a single build (#4) from Aug 2, 2018 at 3:05 AM.
- Project Test Results:**
 - Workspace:** Recent changes and latest test result (no failures).
 - Upstream Projects:** One upstream project, "Compile".
 - Test Result Trend:** A chart showing the count of test results over time, ranging from 0 to 60.
 - Test Result:** 0 failures, 1 skipped. Took 2.2 sec.
 - All Tests:** A table showing test details:

Package	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total (diff)
behavior	74 ms	0	1 +1	1 +1	2 +2
com.wakaleo.gameoflife.domain	0.1 sec	0	0	49 +49	49 +49
com.wakaleo.gameoflife.webtests.controllers	0.72 sec	0	0	7 +7	7 +7

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 screenshot below:

The screenshot shows the Jenkins 'Build Triggers' configuration. The 'Build after other projects are built' checkbox is checked. Under 'Projects to watch', there is a text input field containing 'test'. Below it are three radio buttons: 'Trigger only if build is stable' (selected), 'Trigger even if the build is unstable', and 'Trigger even if the build fails'. There are also three additional checkboxes: 'Build periodically', 'GitHub hook trigger for GITScm polling', and 'Poll SCM', each with a corresponding help icon.

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:

The screenshot shows the Jenkins 'Build' step configuration. It contains a single 'Execute shell' block with the command 'mvn package' entered. There is a link 'See the list of available environment variables' and a 'Advanced...' button.

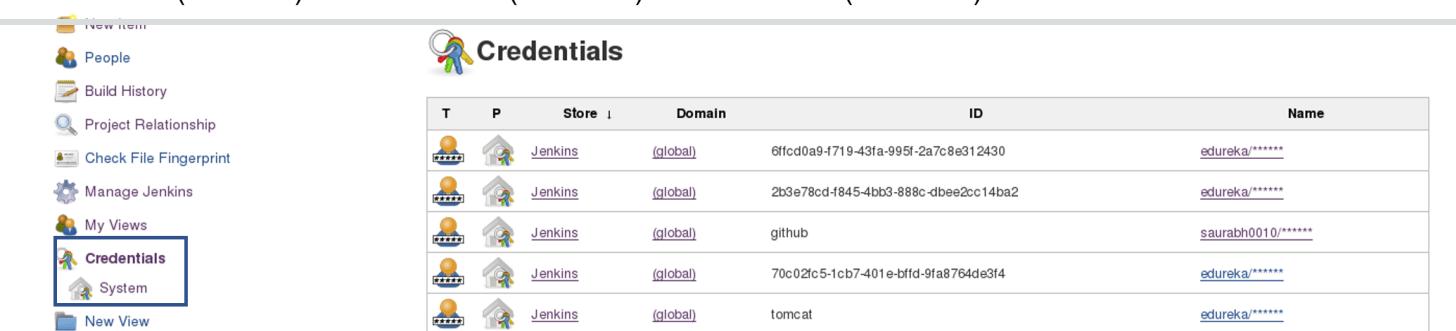
Next step is to deploy this WAR file to the Tomcat server. In the "Post-Build Actions" tab select deploy war/ear to a container. Here, give the path to the war file and give the context path. Consider the below screenshot:

The screenshot shows the Jenkins 'Post-build Actions' configuration. It contains a 'Deploy war/ear to a container' block. The 'WAR/EAR files' field is set to '**/*.war' and the 'Context path' field is set to '/gof'. A 'Containers' section shows a 'Tomcat 7.x' entry with 'Credentials' set to 'edureka/*****' and 'Tomcat URL' set to 'http://localhost:8081'. There is also an 'Add Container' button.

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.

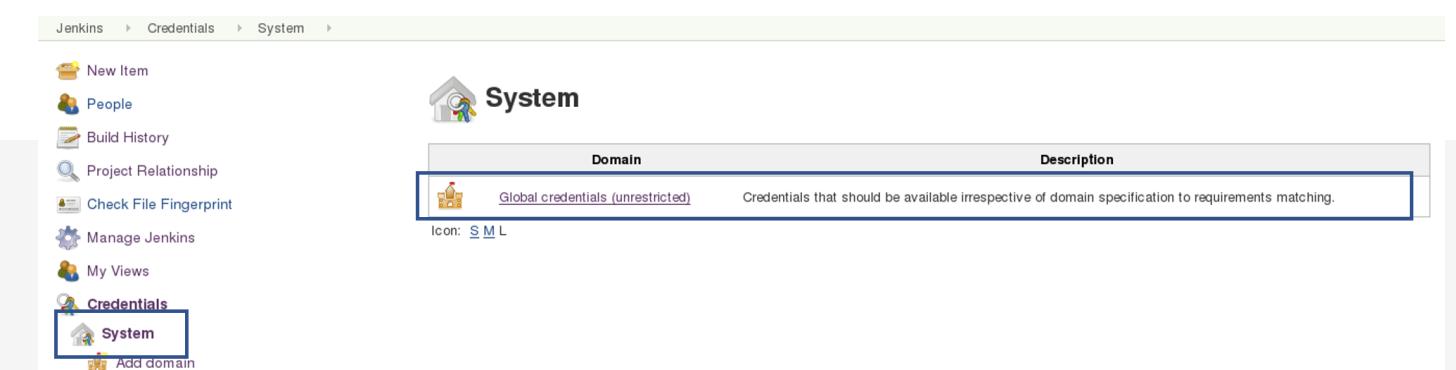
REFCARDZ (/refcardz) **RESEARCH** (/research) **WEBINARS** (/webinars) **ZONES** ▾



The screenshot shows the Jenkins dashboard with the 'Credentials' option selected in the sidebar. The main content area displays a table titled 'Credentials' with columns: T, P, Store, Domain, ID, and Name. The table lists several global Jenkins credentials:

T	P	Store	Domain	ID	Name
jenkins	jenkins	(global)	6ffcd0a9-f719-43fa-995f-2a7c8e312430	edureka*****	
jenkins	jenkins	(global)	2b3e78cd-f845-4bb3-888c-dbee2cc14ba2	edureka*****	
jenkins	jenkins	(global)	github	saurabh0010*****	
jenkins	jenkins	(global)	70c02fc5-1cb7-401e-bffd-9fa8764de3f4	edureka*****	
jenkins	jenkins	(global)	tomcat	edureka*****	

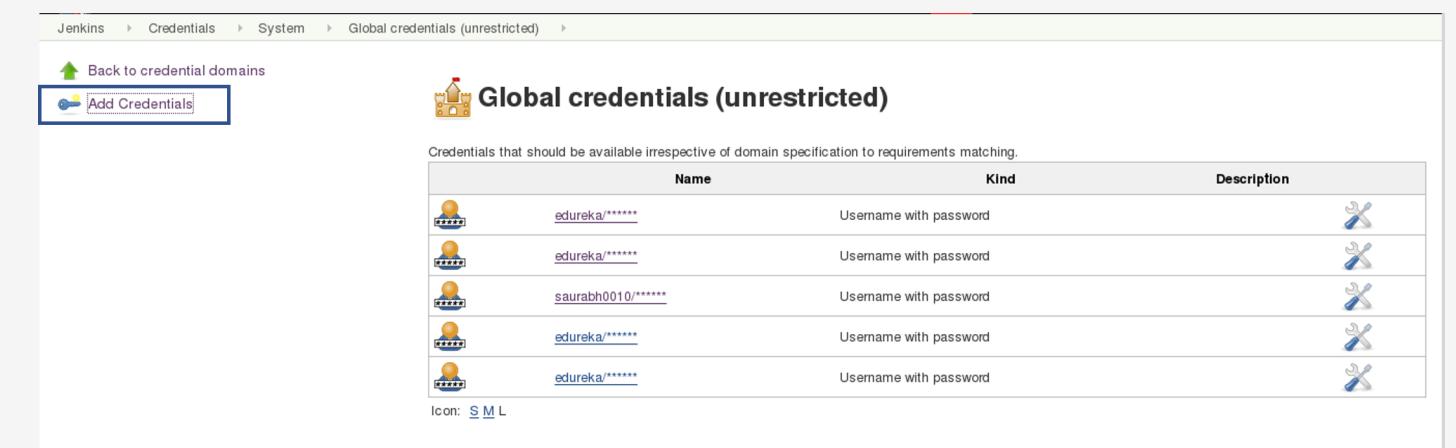
Click on System and select global credentials.



The screenshot shows the Jenkins System configuration page. The 'System' option is selected in the sidebar. The main content area shows a table titled 'Global credentials (unrestricted)' with columns: Domain and Description. One row is listed:

Domain	Description
Global credentials (unrestricted)	Credentials that should be available irrespective of domain specification to requirements matching.

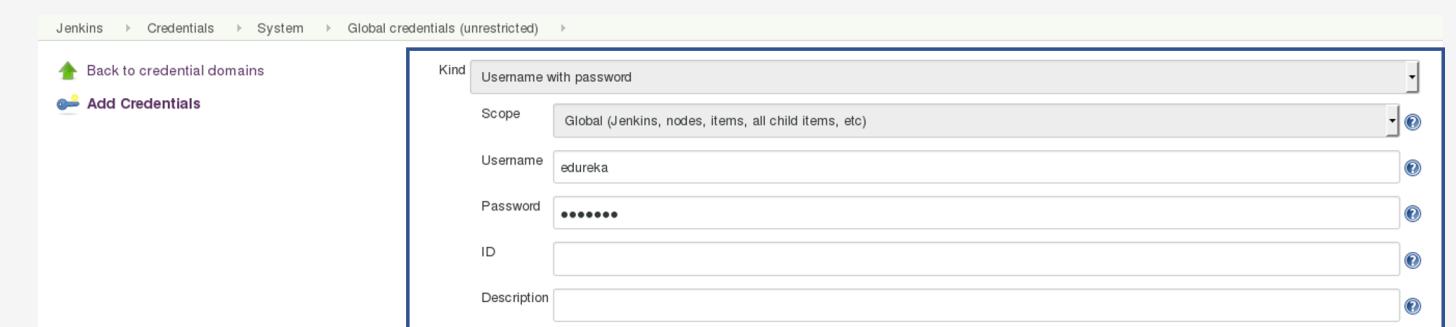
Then you will find an option to add the credentials. Click on it and add credentials.



The screenshot shows the 'Global credentials (unrestricted)' configuration page. The 'Add Credentials' button is highlighted in a blue box. The main content area shows a table with existing credentials and a note: 'Credentials that should be available irrespective of domain specification to requirements matching.'

Name	Kind	Description
edureka*****	Username with password	X
edureka*****	Username with password	X
saurabh0010*****	Username with password	X
edureka*****	Username with password	X
edureka*****	Username with password	X

Add the Tomcat credentials, consider the below screenshot.



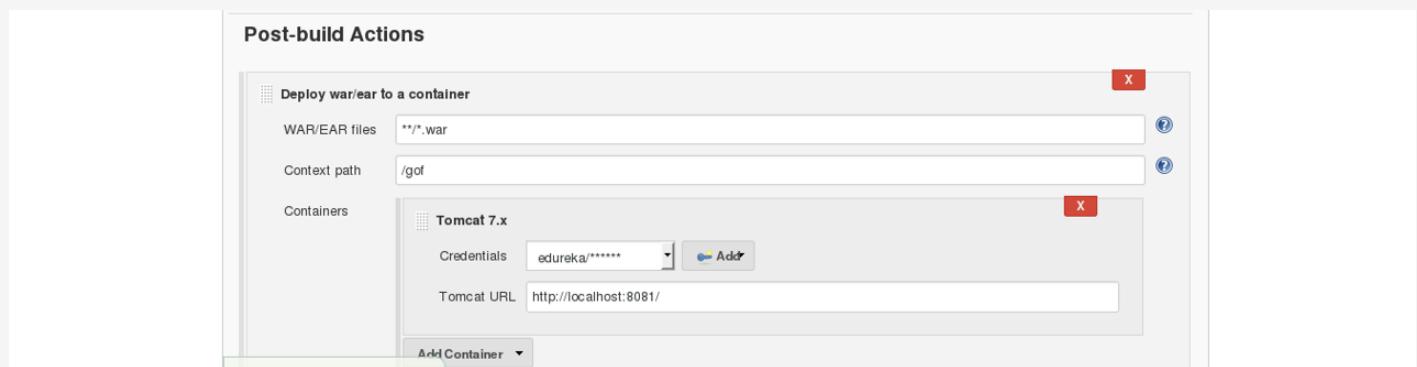
The screenshot shows the 'Add Credentials' dialog for 'Username with password' type. The dialog fields are as follows:

- Kind: Username with password
- Scope: Global (Jenkins, nodes, items, all child items, etc)
- Username: edureka
- Password: *****
- ID: (empty)
- Description: (empty)


[REFCARDZ](#) (/refcardz) [RESEARCH](#) (/research) [WEBINARS](#) (/webinars) [ZONES](#) ▾

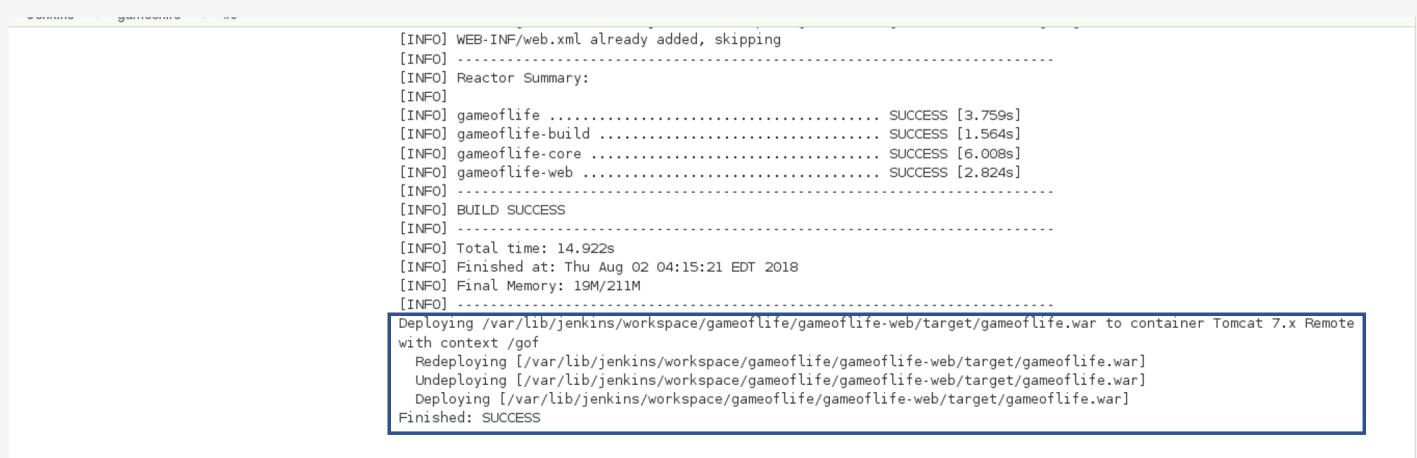
Click on OK.

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



The screenshot shows the 'Post-build Actions' configuration in Jenkins. Under the 'Deploy war/ear to a container' section, the 'WAR/EAR files' field is set to '**/*.war'. The 'Context path' field is set to '/gof'. In the 'Containers' section, there is a single entry for 'Tomcat 7.x'. The 'Credentials' dropdown is set to 'edureka*****'. The 'Tomcat URL' field is set to 'http://localhost:8081/'. There is also an 'Add Container' button.

Click on Save and then select Build Now.



The screenshot shows the Jenkins build log. It displays various build steps and their success status. A specific section highlights the deployment process:

```
[INFO] Deploying /var/lib/jenkins/workspace/gameoflife/gameoflife-web/target/gameoflife.war to container Tomcat 7.x Remote with context /gof
  Redeploying [/var/lib/jenkins/workspace/gameoflife/gameoflife-web/target/gameoflife.war]
  Undeploying [/var/lib/jenkins/workspace/gameoflife/gameoflife-web/target/gameoflife.war]
  Deploying [/var/lib/jenkins/workspace/gameoflife/gameoflife-web/target/gameoflife.war]
Finished: SUCCESS
```

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:



The screenshot shows a web browser window with the URL 'localhost:8081/gof/' in the address bar. The page title is 'Welcome to Conway's Game Of Life!'. The main content area contains a paragraph about the Game of Life and its rules. At the bottom left, there is a 'New Game' button.

This is a really cool web version of Conway's famous Game Of Life. The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway way back in 1970.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any dead cell with exactly three live neighbours becomes a live cell.

New Game



1 Link - http://localhost:8081/gof (/)

A DEVADA MEDIA PROPERTY

REFCARDZ (/refcardz) RESEARCH (/research) WEBINARS (/webinars) ZONES ▾

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

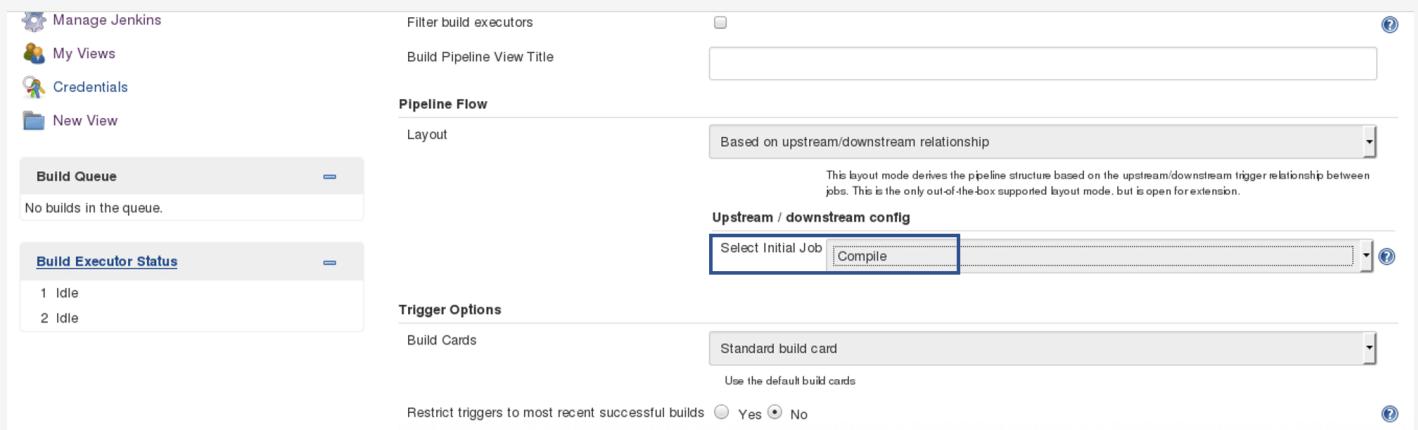
Now, to create a pipeline view, consider the below screenshot:



The screenshot shows the Jenkins Pipeline View configuration page. At the top, there are links for 'New Item' and 'People'. Below that, a search bar contains 'Game of Life'. A blue plus icon is located at the end of the search bar. On the left, there's a sidebar with links for 'Manage Jenkins', 'My Views', 'Credentials', and 'New View'. Under 'Build Queue', it says 'No builds in the queue.' Under 'Build Executor Status', there are two entries: '1 Idle' and '2 Idle'. The main configuration area has sections for 'Pipeline Flow' (Layout: 'Based on upstream/downstream relationship'), 'Upstream / downstream config' (Select Initial Job: 'Compile'), and 'Trigger Options' (Build Cards: 'Standard build card', 'Use the default build cards', 'Restrict triggers to most recent successful builds' with radio buttons for 'Yes' and 'No').

Click on the plus icon to create a new view.

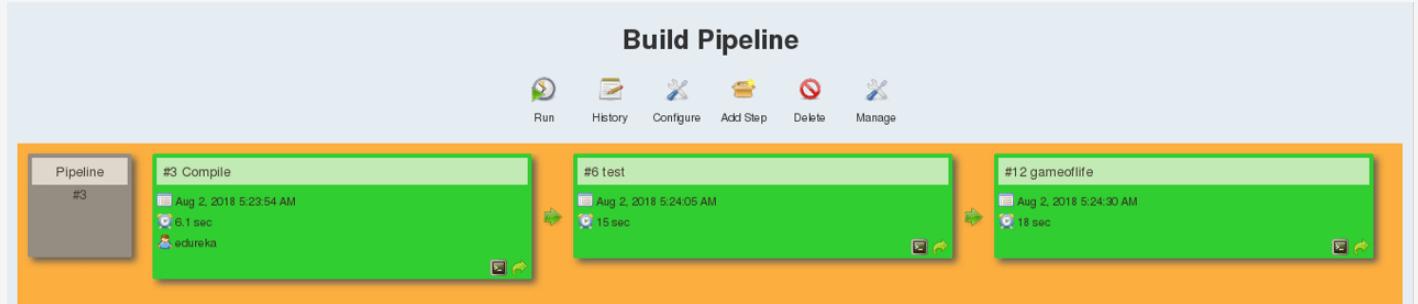
Configure the pipeline the way you want:



This screenshot shows the Jenkins Pipeline View configuration page again. The 'New View' link in the sidebar is highlighted. The 'Upstream / downstream config' section has 'Select Initial Job' set to 'Compile'. The 'Trigger Options' section shows 'Standard build card' selected. The 'Restrict triggers to most recent successful builds' section has 'No' selected.

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.



The screenshot shows the Jenkins Build Pipeline interface. It displays three green cards representing pipeline stages: '#3 Compile' (Aug 2, 2018 5:23:54 AM, 6.1 sec, eduska), '#6 test' (Aug 2, 2018 5:24:05 AM, 15 sec), and '#12 gameoflife' (Aug 2, 2018 5:24:30 AM, 18 sec). Above the cards is a toolbar with icons for Run, History, Configure, Add Step, Delete, and Manage. The cards are connected by arrows, indicating a sequential flow from compile to test to deployment.

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



Published at DZone with permission of Saurabh K, DZone MVB Contributors original article
(https://www.edureka.co/blog/continuous-delivery/)



MVB Contributors

Original article

(/search)

REPOARDS expressed RESEARCH contributions and WEBINARS (./webinars) ZONES ▾

ABOUT US

[About DZone \(/pages/about\)](/pages/about)

[Send feedback \(mailto:support@dzone.com\)](mailto:support@dzone.com)

[Careers \(https://devada.com/careers/\)](https://devada.com/careers/)

ADVERTISE

[Developer Marketing Blog \(https://devada.com/blog/developer-marketing\)](https://devada.com/blog/developer-marketing)

[Advertise with DZone \(/pages/advertise\)](/pages/advertise)

+1 (919) 238-7100 (tel:+19192387100)

CONTRIBUTE ON DZONE

[MVB Program \(/pages/mvb\)](/pages/mvb)

[Zone Leader Program \(/pages/zoneleader\)](/pages/zoneleader)

[Become a Contributor \(/pages/contribute\)](/pages/contribute)

[Visit the Writers' Zone \(/writers-zone\)](/writers-zone)

LEGAL

[Terms of Service \(/pages/tos\)](/pages/tos)

[Privacy Policy \(/pages/privacy\)](/pages/privacy)

CONTACT US

600 Park Offices Drive

Suite 150

Research Triangle Park, NC 27709

[support@dzone.com \(mailto:support@dzone.com\)](mailto:support@dzone.com)

+1 (919) 678-0300 (tel:+19196780300)

Let's be friends:

in



(<https://www.linkedin.com/company/devada->
<https://www.linkedin.com/company/DZoneInc>)

DZone.com is powered by

AnswerHub™ (<https://devada.com/answerhub/>)
A DEVADA SOFTWARE PRODUCT