

Jenkins Pipelines

Raúl Estrada

Octubre 2020

1. Overview

In this article, we're going to showcase the usage of pipelines through an example of continuous delivery using [Jenkins](#).

We're going to build a simple, yet quite useful pipeline, for our sample project:

- Compilation
- Simple static analysis (parallel with compilation)
- Unit tests
- Integration tests (parallel with unit tests)
- Deployment

2. Setting up Jenkins

First of all, we need to download the latest stable version of [Jenkins](#) (2.73.3 at the point of writing this article).

Let's navigate to the folder where our file is and run it using the `java -jar jenkins.war` command. **Keep in mind that we can't use Jenkins without an initial users setup.**

After unlocking Jenkins by using the initial admin generated password, we must fill profile information of the first admin user and be sure to install all recommended plugins.

Getting Started

Getting Started


✓ Folders Plugin	✓ OWASP Markup Formatter Plugin	✓ Build Timeout	✓ Credentials Binding Plugin	Gradle Plugin ** Pipeline: Milestone Step ** JavaScript GUI Lib: jQuery bundles (jQuery and jQuery UI) plugin ** Jackson 2 API Plugin ** JavaScript GUI Lib: ACE Editor bundle plugin ** Pipeline: SCM Step ** Pipeline: Groovy ** Pipeline: Input Step ** Pipeline: Stage Step ** Pipeline Graph Analysis Plugin ** Pipeline: REST API Plugin ** JavaScript GUI Lib: Handlebars bundle plugin ** JavaScript GUI Lib: Moment.js bundle plugin Pipeline: Stage View Plugin ** Pipeline: Build Step ** Pipeline: Model API ** - required dependency
✓ Timestampers	✓ Workspace Cleanup Plugin	✓ Ant Plugin	✓ Gradle Plugin	
🔄 Pipeline	🔄 GitHub Branch Source Plugin	🔄 Pipeline: GitHub Groovy Libraries	✓ Pipeline: Stage View Plugin	
🔄 Git plugin	🔄 Subversion Plug-in	🔄 SSH Slaves plugin	○ Matrix Authorization Strategy Plugin	
🔄 PAM Authentication plugin	🔄 LDAP Plugin	🔄 Email Extension Plugin	🔄 Mailer Plugin	

Now we have a fresh installation of Jenkins ready to be used.


 **Jenkins**


[?](#) [admin](#) | [log out](#)

Jenkins > [ENABLE AUTO REFRESH](#)

 [New Item](#)


 [People](#)

 [Build History](#)


 [Manage Jenkins](#)

 [My Views](#)

 [Credentials](#)

Build Queue 

No builds in the queue.

Build Executor Status 

1 Idle

2 Idle

Welcome to Jenkins!

Please [create new jobs](#) to get started.

 [add description](#)

3. Pipelines

Jenkins 2 comes with a great feature called *Pipelines*, which is very extensible when we need to define a continuous integration environment for a project.

A Pipeline is another way of defining some Jenkins steps using code, and automate the process of deploying software.

It's using a **Domain Specific Language(DSL)** with two different syntaxes:

- Declarative Pipeline
- Scripted Pipeline

In our examples, we're going to use **the *Scripted Pipeline* which is following a more imperative programming model built with Groovy.**

Let's go through some characteristics of the *Pipeline* plugin:

- pipelines are written into a text file and treated as code; this means they can be added to version control and modified later on
- they will remain after restarts of the Jenkins server
- we can optionally pause pipelines
- they support complex requirements such as performing work in parallel
- the Pipeline plugin can also be extended or integrated with other plugins

In other words, setting up a Pipeline project means writing a script that will sequentially apply some steps of the process we want to accomplish.

To start using pipelines we have to install the Pipeline plugin that allows composing simple and complex automation.

We can optionally have the Pipeline Stage View one too so that when we run a build, we'll see all the stages we've configured.

4. A Quick Example

For our example, we'll use a small Spring Boot application. We'll then create a pipeline which clones the project, builds it and runs several tests, then runs the application.

Let's install the *Checkstyle*, *Static Analysis Collector* and *JUnit* plugins, which are respectively useful to collect *Checkstyle* results, build a combined analysis graph of the test reports and illustrate successfully executed and failed tests.

First, let's understand the reason of Checkstyle here: it's a development tool that helps programmers write better Java code following accepted and well-known standards.

Static Analysis Collector is an add-on which collects different analysis outputs and prints the results in a combined trend graph. Additionally, the plug-in provides health reporting and build stability based on these grouped results.

Finally, the *JUnit* plugin provides a publisher that consumes XML test reports generated during the builds and outputs detailed and meaningful information relative to a project's tests.

We'll also configure *Checkstyle* in our application's *pom.xml*:

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-checkstyle-plugin</artifactId>
4   <version>2.17</version>
5 </plugin>
```

5. Creating a Pipeline Script

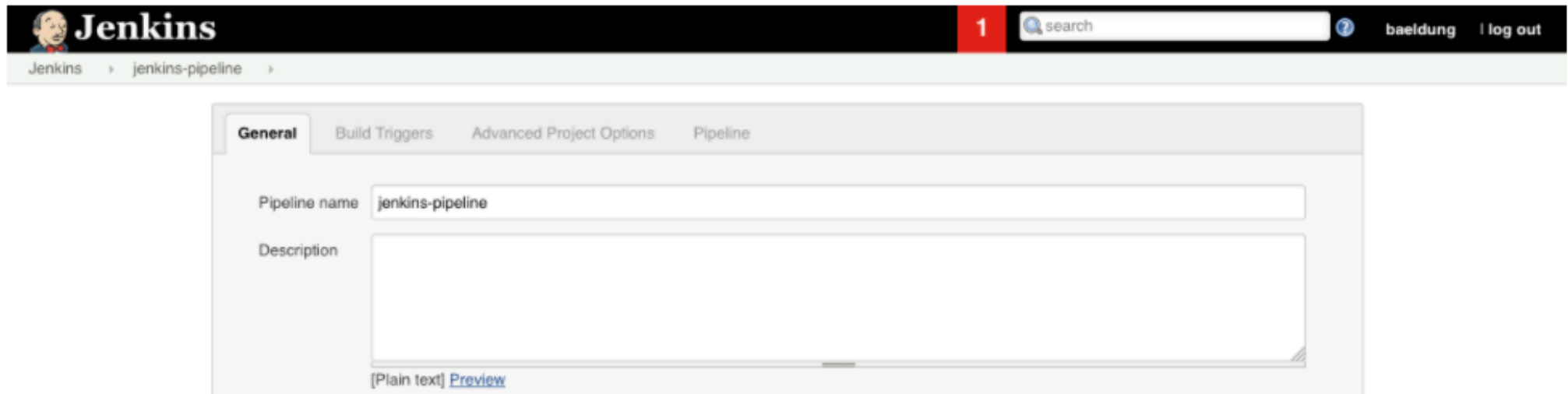
First, we need to create a new Jenkins job. Let's be sure to select *Pipeline* as the type before hitting the OK button as described in this screenshot:

The screenshot shows the Jenkins web interface for creating a new item. At the top, the Jenkins logo and navigation links are visible. The main form has a header 'Enter an item name' and a text input field containing 'jenkins-pipeline'. Below the input field, there are five job type options, each with an icon and a description:

- Freestyle project**: This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Maven project**: Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- Pipeline**: Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type. (This option is selected with a blue border)
- Multi-configuration project**: Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Folder**: Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

At the bottom left of the form, there is a blue 'OK' button. The bottom of the screenshot is partially cut off, showing the start of another option 'GitHub Organization'.

The next screen allows us to fill in more details of the different steps of our Jenkins job, such as the *description*, *triggers*, some *advanced project options*:



The image shows the Jenkins web interface for configuring a job named 'jenkins-pipeline'. The top navigation bar is black with the Jenkins logo on the left, a red square with the number '1' in the center, and a search bar on the right. Below the navigation bar, a breadcrumb trail shows 'Jenkins > jenkins-pipeline >'. The main content area has a light gray background and contains a tabbed interface with four tabs: 'General' (selected), 'Build Triggers', 'Advanced Project Options', and 'Pipeline'. The 'General' tab is active and shows two input fields: 'Pipeline name' with the value 'jenkins-pipeline' and 'Description' with a large empty text area. At the bottom of the 'Description' field, there is a small text label '[Plain text]' and a blue link labeled 'Preview'.

Jenkins 1 search ? baeldung | log out

Jenkins > jenkins-pipeline >

General Build Triggers Advanced Project Options Pipeline

Pipeline name jenkins-pipeline

Description

[Plain text] [Preview](#)

Let's dive into the main and most important part of this kind of job by clicking on the *Pipeline* tab.

Then, for the definition select *Pipeline script* and check *Use Groovy Sandbox*.

Here is the working script for a Unix environment:

```
1  node {
2      stage 'Clone the project'
3      git 'https://github.com/eugenp/tutorials.git'
4
5      dir('spring-jenkins-pipeline') {
6          stage("Compilation and Analysis") {
7              parallel 'Compilation': {
8                  sh "./mvnw clean install -DskipTests"
9              }, 'Static Analysis': {
10                 stage("Checkstyle") {
11                     sh "./mvnw checkstyle:checkstyle"
12
13                     step([$class: 'CheckStylePublisher',
14                         canRunOnFailed: true,
15                         defaultEncoding: '',
16                         healthy: '100',
17                         pattern: '**/target/checkstyle-result.xml',
18                         unhealthy: '90',
19                         useStableBuildAsReference: true
20                     ])
21                 }
22             }
23         }
24     }
```

```
25 stage("Tests and Deployment") {
26     parallel 'Unit tests': {
27         stage("Runing unit tests") {
28             try {
29                 sh "./mvnw test -Punit"
30             } catch(err) {
31                 step([$class: 'JUnitResultArchiver', testResults:
32                     '**/target/surefire-reports/TEST-*UnitTest.xml'])
33                 throw err
34             }
35             step([$class: 'JUnitResultArchiver', testResults:
36                 '**/target/surefire-reports/TEST-*UnitTest.xml'])
37         }
38     }, 'Integration tests': {
39         stage("Runing integration tests") {
40             try {
41                 sh "./mvnw test -Pintegration"
42             } catch(err) {
43                 step([$class: 'JUnitResultArchiver', testResults:
44                     '**/target/surefire-reports/TEST-'
45                     + '*IntegrationTest.xml'])
46                 throw err
47             }
48             step([$class: 'JUnitResultArchiver', testResults:
49                 '**/target/surefire-reports/TEST-'
50                 + '*IntegrationTest.xml'])
51         }
52     }
53 }
```

```
54     stage("Staging") {
55         sh "pid=\$(lsof -i:8989 -t); kill -TERM \$pid "
56           + "|| kill -KILL \$pid"
57         withEnv(['JENKINS_NODE_COOKIE=dontkill']) {
58             sh 'nohup ./mvnw spring-boot:run -Dserver.port=8989 &'
59         }
60     }
61 }
62 }
63 }
```

First, we're cloning the repository from GitHub, then changing the directory to our project, which is called *spring-jenkins-pipeline*.

Next, we compiled the project and apply *Checkstyle* analysis in a parallel way.

The following step represents a parallel execution of unit tests and integration tests, then deployment of the app.

Parallelism is used to optimize the pipeline, and have the job runs faster. It's a best practice in Jenkins to simultaneously run some independent actions that can take a lot of time.

For example, in a real-world project, we usually have a lot of unit and integration tests that can take longer.

Note that if any test failed the BUILD will be marked as FAILED too and the deployment will not occur.

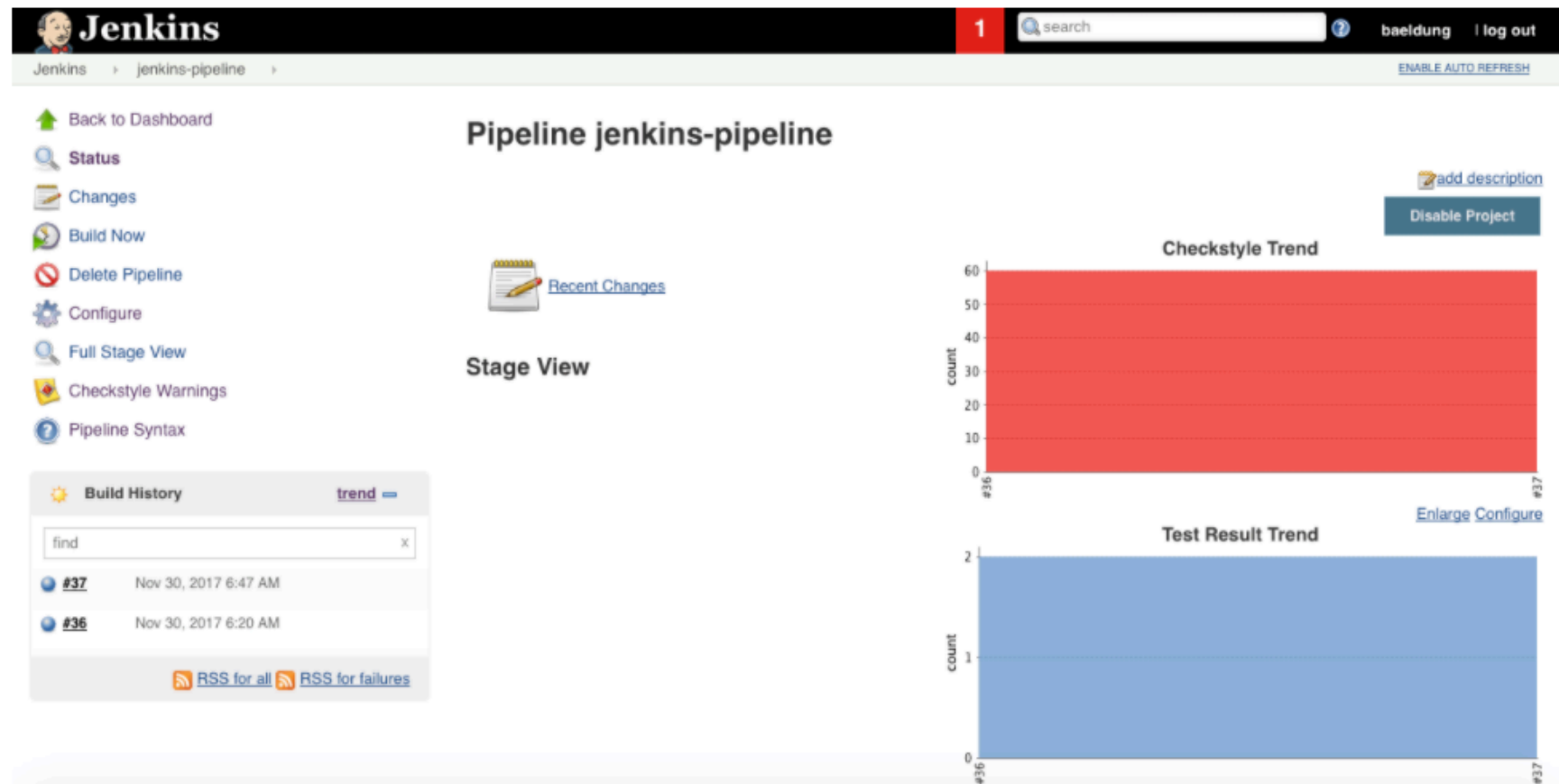
Also, we're using *JENKINS_NODE_COOKIE* to prevent immediate shut down of our application when the pipeline reaches the end.

To see a more general script working on other different systems, check out the [GitHub repository](#).

6. Analysis Report

After creating the job, we'll save our script and hit *Build Now* on the project home of our Jenkins dashboard.

Here's an overview of the builds:



A little further down we'll find the stage view of the pipeline, with the result of each stage:


[\(just show failures\)](#) [enlarge](#)

		Clone the project	Compilation and Analysis	Checkstyle	Tests and Deployment	Runing unit tests	Runing integration tests	Staging
Average stage times: (Average <u>full</u> run time: ~1min 12s)		4s	169ms	23s	69ms	20s	41s	42s
#37	Nov 30 06:47 No Changes	5s	216ms	23s	69ms	21s	39s	40s
#36	Nov 30 06:20 No Changes	4s	122ms	23s	70ms	20s	44s	44s


Each output is accessible when hovering over a stage cell and clicking the *Logs* button to see the log messages printed in that step.


We can also find more details of the code analysis. Let's click on the desired build from the *Build History* on the right menu and hit *Checkstyle Warnings*.

Here we see 60 high priority warnings browsable by clicking:


 **Jenkins**


1


 search


 baeldung | [log out](#)


Jenkins > jenkins-pipeline > #37 > Checkstyle Warnings


 [Back to Project](#)


 [Status](#)


 [Changes](#)


 [Console Output](#)


 [Edit Build Information](#)


 [Delete Build](#)


 [Git Build Data](#)


 [No Tags](#)

 **Checkstyle Warnings**

 [Test Result](#)

 [Replay](#)

 [Pipeline Steps](#)

 [Previous Build](#)

CheckStyle Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
60	0	0

Summary

Total	High Priority	Normal Priority	Low Priority
60	60	0	0

Details

Packages | Files | Categories | Types | Warnings | Origin | Details

Package	Total	Distribution
com.baeldung	8	<div></div>
com.baeldung.domain	49	<div></div>
com.baeldung.repository	2	<div></div>
src/main/resources	1	<div></div>
Total	60	

Page generated: Nov 30, 2017 7:00:47 AM WAT [REST API](#) [Jenkins ver. 2.73.3](#)

The *Details* tab displays pieces of information that highlight warnings and enable the developer to understand the causes behind them.

In the same way, the full test report is accessible by clicking on *Test Result* link. Let's see the results of the *com.baeldung* package:

The screenshot shows the Jenkins web interface. At the top, the Jenkins logo is on the left, a red status bar with the number '1' in the center, and a search bar on the right. Below the status bar, a breadcrumb trail reads 'Jenkins > jenkins-pipeline > #37 > Test Results > com.baeldung'. On the far right of this bar is a link to 'ENABLE AUTO REFRESH'. A left-hand sidebar contains several links: 'History', 'Git Build Data', 'No Tags', 'Checkstyle Warnings', 'Test Result' (which is highlighted), 'Replay', 'Pipeline Steps', and 'Previous Build'. The main content area is titled 'Test Result : com.baeldung'. Below the title, it says '0 failures (±0)' and shows a blue progress bar. To the right of the bar, it indicates '2 tests (±0)', 'Took 18 sec.', and a link to 'add description'. Below this, the section 'All Tests' contains a table with the following data:

Class	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total (diff)
SomeIntegrationTest	17 sec	0	0	1	1
SomeUnitTest	0.35 sec	0	0	1	1

Here we can see each test file with its duration and status.

7. Conclusion

In this article, we set up a simple continuous delivery environment to run and show static code analysis and test report in Jenkins via a *Pipeline* job.