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:

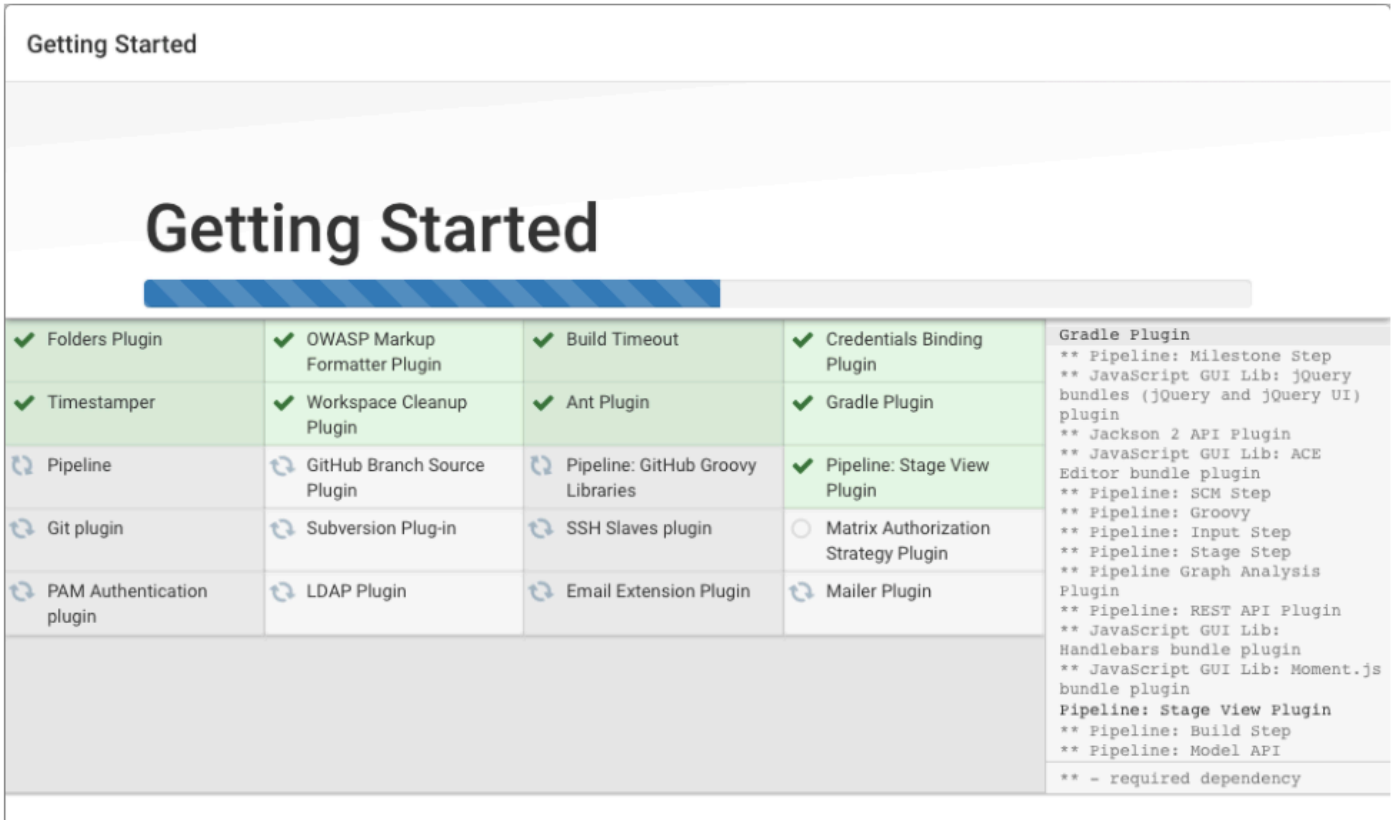
- Compilation
- Simple static analysis (parallel with compilation)
- Unit tests
- Integration tests (parallel with unit tests)
- Deployment

2. Setting up Jenkins

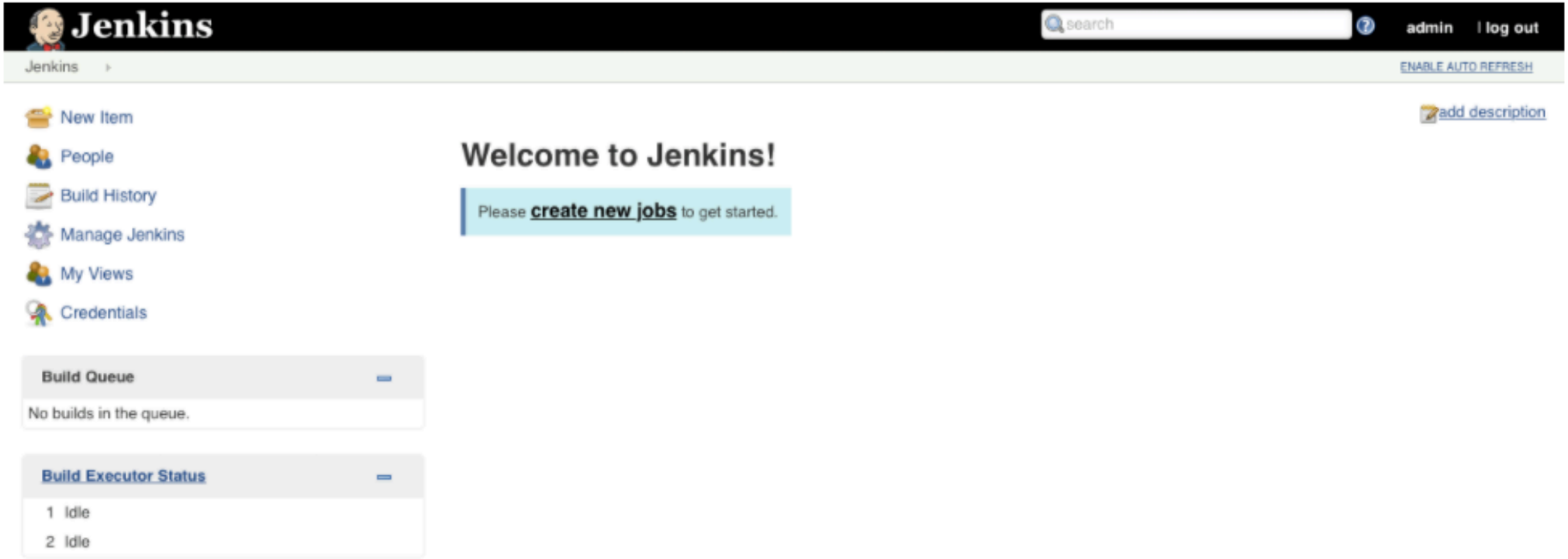
First of all, we need to download the latest stable version of Jenkins (2.73.3 at the point of writing this article).

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.



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



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.

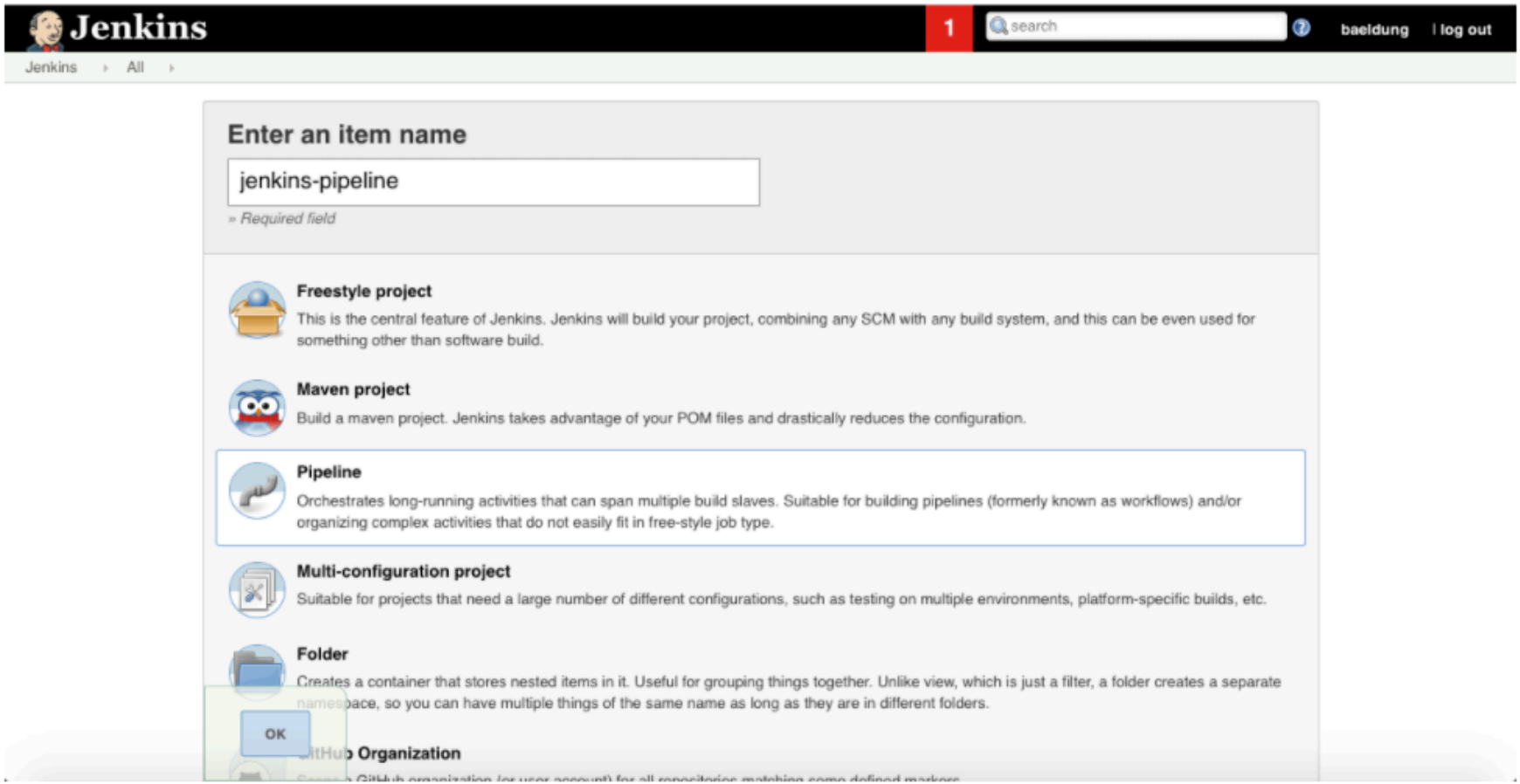
Finally, the *JUnit* plugin provides a publisher that consumes XML test reports generated during the builds and outputs detailed and meaningful information relative to a project's tests.

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

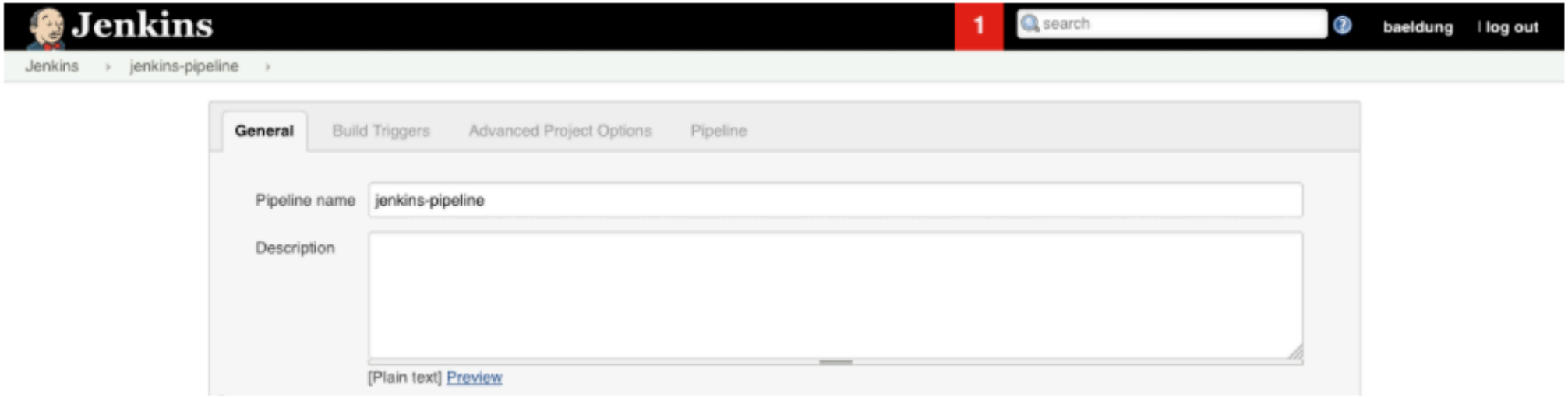
```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.17</version>
</plugin>
```

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 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:*



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:

```
node {
    stage 'Clone the project'
    git 'https://github.com/eugenp/tutorials.git'

    dir('spring-jenkins-pipeline') {
        stage("Compilation and Analysis") {
            parallel 'Compilation': {
                sh "./mvnw clean install -DskipTests"
            }, 'Static Analysis': {
                stage("Checkstyle") {
                    sh "./mvnw checkstyle:checkstyle"

                    step([$class: 'CheckStylePublisher',
                      canRunOnFailed: true,
                      defaultEncoding: '',
                      healthy: '100',
                      pattern: '**/target/checkstyle-result.xml',
                      unHealthy: '90',
                      useStableBuildAsReference: true
                    ])
                }
            }
        }
```

```
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              }
```

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

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.

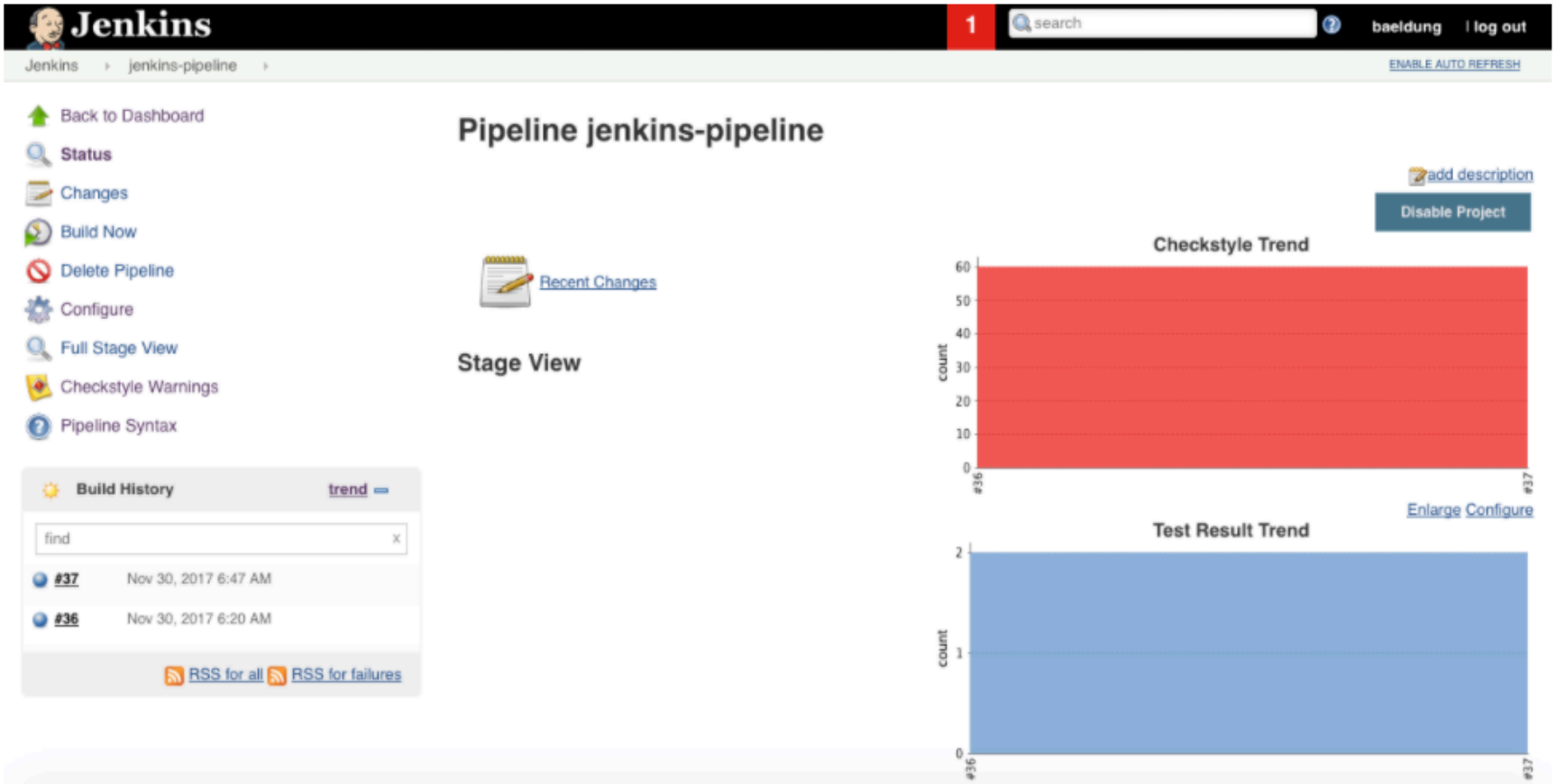
Also, we're using *JENKINS_NODE_COOKIE* to prevent immediate shut down of our application when the pipeline reaches the end.

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 full 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

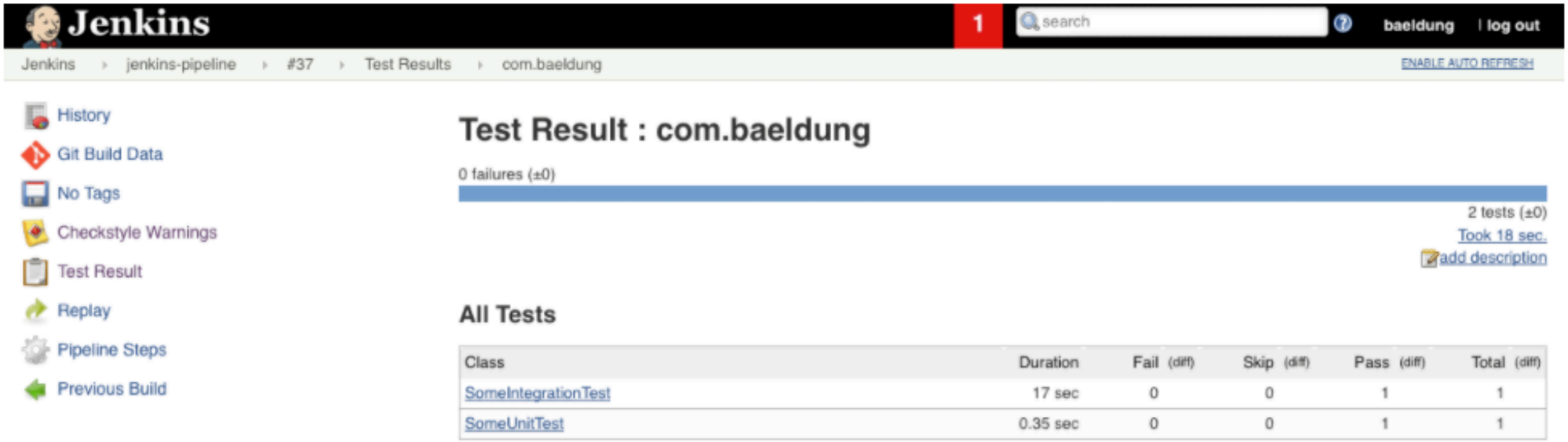
Packages | Files | Categories | Types | Warnings | Origin | Details

Package	Total	Distribution
com.baeldung	8	
com.baeldung.domain	49	
com.baeldung.repository	2	
src/main/resources	1	
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:



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.