

Commit Pipeline

Raúl Estrada

Octubre 2020

Commit pipeline

The most basic Continuous Integration process is called a commit pipeline. This classic phase, as its name says, starts with a commit (or push in Git) to the main repository and results in a report about the build success or failure. Since it runs after each change in the code, the build should take no more than 5 minutes and should consume a reasonable amount of resources. The commit phase is always the starting point of the Continuous Delivery process, and it provides the most important feedback cycle in the development process, constant information if the code is in a healthy state.

The commit phase works as follows. A developer checks in the code to the repository, the Continuous Integration server detects the change, and the build starts. The most fundamental commit pipeline contains three stages:

- **Checkout:** This stage downloads the source code from the repository
- **Compile:** This stage compiles the source code
- **Unit test:** This stage runs a suite of unit tests

Let's create a sample project and see how to implement the commit pipeline.

Checkout

Checking out code from the repository is always the first operation in any pipeline. In order to see this, we need to have a repository. Then, we will be able to create a pipeline.

Creating a GitHub repository

Creating a repository on the GitHub server takes just a few steps:

1. Go to the <https://github.com/> page.
2. Create an account if you don't have one yet.
3. Click on **New repository**.
4. Give it a name, `calculator`.
5. Tick **Initialize this repository with a README**.
6. Click on **Create repository**.

Creating a checkout stage

We can create a new pipeline called `calculator` and, as **Pipeline script**, put the code with a stage called **Checkout**:

```
pipeline {
  agent any
  stages {
    stage("Checkout") {
      steps {
        git url: 'https://github.com/leszko/calculator.git'
      }
    }
  }
}
```

The pipeline can be executed on any of the agents, and its only step does nothing more than downloading code from the repository. We can click on **Build Now** and see if it was executed successfully.

Compile

In order to compile a project, we need to:

1. Create a project with the source code.
2. Push it to the repository.
3. Add the **Compile** stage to the pipeline.

The simplest way to create a Spring Boot project is to perform the following steps:

1. Go to the `http://start.spring.io/` page.
2. Select **Gradle project** instead of **Maven project** (you can also leave Maven if you prefer it to Gradle).
3. Fill **Group** and **Artifact** (for example, `com.leszko` and `calculator`).
4. Add **Web** to **Dependencies**.
5. Click on **Generate Project**.
6. The generated skeleton project should be downloaded (the `calculator.zip` file).

Pushing code to GitHub

We will use the Git tool to perform the `commit` and `push` operations:



In order to run the `git` command, you need to have the Git toolkit installed (it can be downloaded from <https://git-scm.com/downloads>).

Let's first clone the repository to the filesystem:

```
$ git clone https://github.com/leszko/calculator.git
```

Extract the project downloaded from <http://start.spring.io/> into the directory created by Git.

As a result, the calculator directory should have the following files:

```
$ ls -a
. .. build.gradle .git .gitignore gradle gradlew gradlew.bat README.md src
```



In order to perform the Gradle operations locally, you need to have Java JDK installed (in Ubuntu, you can do it by executing `sudo apt-get install -y default-jdk`).

We can compile the project locally using the following code:

```
$ ./gradlew compileJava
```

In the case of Maven, you can run `./mvnw compile`. Both Gradle and Maven compile the Java classes located in the `src` directory.

Now, we can commit and push to the GitHub repository:

```
$ git add .  
$ git commit -m "Add Spring Boot skeleton"  
$ git push -u origin master
```



After running the `git push` command, you will be prompted to enter the GitHub credentials (username and password).

The code is already in the GitHub repository. If you want to check it, you can go to the GitHub page and see the files.

Creating a compile stage

We can add a `Compile` stage to the pipeline using the following code:

```
stage("Compile") {  
    steps {  
        sh "./gradlew compileJava"  
    }  
}
```

Note that we used exactly the same command locally and in the Jenkins pipeline, which is a very good sign because the local development process is consistent with the Continuous Integration environment. After running the build, you should see two green boxes. You can also check that the project was compiled correctly in the console log.

Unit test

It's time to add the last stage that is Unit test, which checks if our code does what we expect it to do. We have to:

- Add the source code for the calculator logic
- Write unit test for the code
- Add a stage to execute the unit test

Let's create a unit test in the file

`src/test/java/com/leszko/calculator/CalculatorTest.java`:

```
package com.leszko.calculator;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalculatorTest {
    private Calculator calculator = new Calculator();

    @Test
    public void testSum() {
        assertEquals(5, calculator.sum(2, 3));
    }
}
```

We can run the test locally using the `./gradlew test` command. Then, let's commit the code and push it to the repository:

```
$ git add .  
$ git commit -m "Add sum logic, controller and unit test"  
$ git push
```

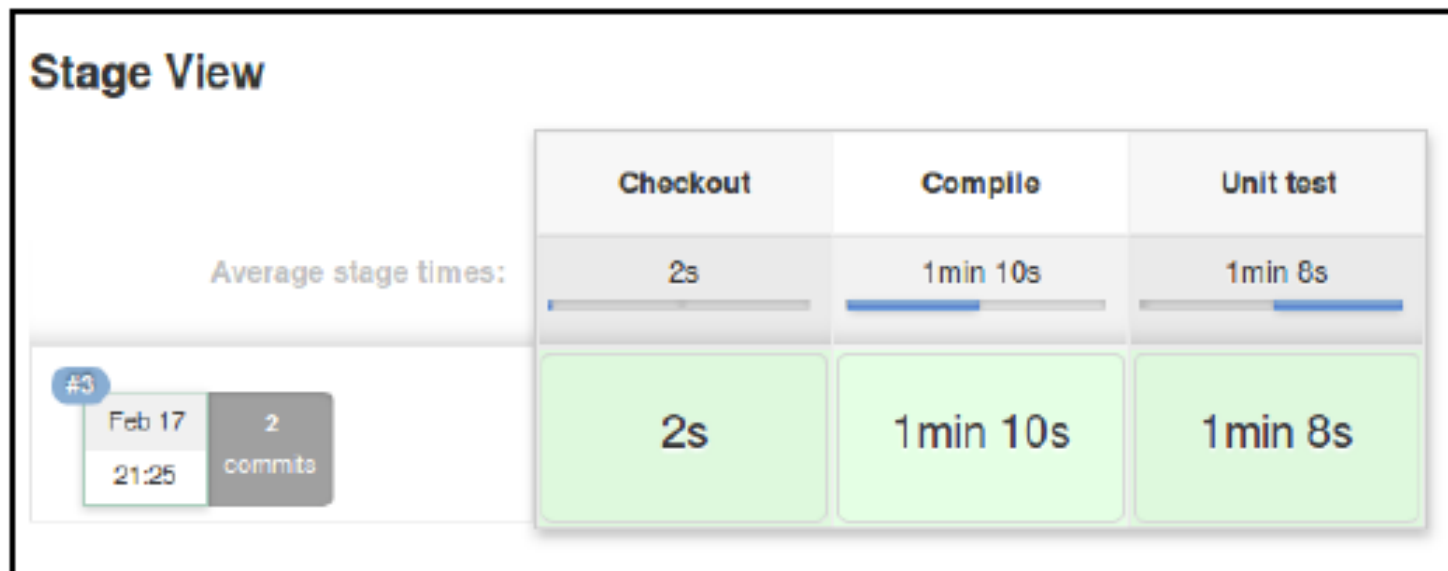
Creating a unit test stage

Now, we can add a Unit test stage to the pipeline:

```
stage("Unit test") {  
    steps {  
        sh "./gradlew test"  
    }  
}
```

In the case of Maven, we would have to use `./mvnw test`.

When we build the pipeline again, we should see three boxes, which means that we've completed the Continuous Integration pipeline:



Let's create a file called `Jenkinsfile` in the root directory of our project:

```
pipeline {
  agent any
  stages {
    stage("Compile") {
      steps {
        sh "./gradlew compileJava"
      }
    }
    stage("Unit test") {
      steps {
        sh "./gradlew test"
      }
    }
  }
}
```

- In case of Jenkins' failure, the pipeline definition is not lost (because it's stored in the code repository, not in Jenkins)
- The history of the pipeline changes is stored
- Pipeline changes go through the standard code development process (for example, they are subjected to code reviews)
- Access to the pipeline changes is restricted exactly in the same way as the access to the source code

We can now commit the added files and push to the GitHub repository:

```
$ git add .  
$ git commit -m "Add sum Jenkinsfile"  
$ git push
```

Running pipeline from Jenkinsfile

When `Jenkinsfile` is in the repository, then all we have to do is to open the pipeline configuration and in the `Pipeline` section:

- Change **Definition** from `Pipeline script` to `Pipeline script from SCM`
- Select **Git** in **SCM**
- Put `https://github.com/leszko/calculator.git` in **Repository URL**

Code coverage

Think about the following scenario: you have a well-configured Continuous Integration process; however, nobody in your project writes unit tests. It passes all the builds, but it doesn't mean that the code is working as expected. What to do then? How to ensure that the code is tested?

The solution is to add the code coverage tool that runs all tests and verifies which parts of the code have been executed. Then, it creates a report showing not-tested sections. Moreover, we can make the build fail when there is too much untested code.

There are a lot of tools available to perform the test coverage analysis; for Java, the most popular are JaCoCo, Clover, and Cobertura.

Let's use JaCoCo and show how the coverage check works in practice. In order to do this, we need to perform the following steps:

1. Add JaCoCo to the Gradle configuration.
2. Add the code coverage stage to the pipeline.
3. Optionally, publish JaCoCo reports in Jenkins.

Adding JaCoCo to Gradle

In order to run JaCoCo from Gradle, we need to add the `jacoco` plugin to the `build.gradle` file by adding the following line in the plugin section:

```
apply plugin: "jacoco"
```

Next, if we would like to make the Gradle fail in case of too low code coverage, we can add the following configuration to the `build.gradle` file as well:

```
jacocoTestCoverageVerification {  
    violationRules {  
        rule {  
            limit {  
                minimum = 0.2  
            }  
        }  
    }  
}
```








This configuration sets the minimum code coverage to 20%. We can run it with the following command:

```
$ ./gradlew test jacocoTestCoverageVerification
```

The command checks if the code coverage is at least 20%. You can play with the minimum value to see the level at which the build fails. We can also generate a test coverage report using the following command:

```
$ ./gradlew test jacocoTestReport
```

You can also have a look at the full coverage report in the `build/reports/jacoco/test/html/index.html` file:

com.leszko.calculator			
Element	Missed Instructions	Cov.	Missed Branches
 CalculatorController		25%	
 CalculatorApplication		38%	
 Calculator		100%	
Total	14 of 27	48%	0 of 0

Adding a code coverage stage

Adding a code coverage stage to the pipeline is as simple as the previous stages:

```
stage("Code coverage") {  
    steps {  
        sh "./gradlew jacocoTestReport"  
        sh "./gradlew jacocoTestCoverageVerification"  
    }  
}
```

After adding this stage, if anyone commits code that is not well-covered with tests, the build will fail.

Publishing the code coverage report

When the coverage is low and the pipeline fails, it would be useful to look at the code coverage report and find what parts are not yet covered with tests. We could run Gradle locally and generate the coverage report; however, it is more convenient if Jenkins shows the report for us.

In order to publish the code coverage report in Jenkins, we need the following stage definition:

```
stage("Code coverage") {
    steps {
        sh "./gradlew jacocoTestReport"
        publishHTML (target: [
            reportDir: 'build/reports/jacoco/test/html',
            reportFiles: 'index.html',
            reportName: "JaCoCo Report"
        ])
        sh "./gradlew jacocoTestCoverageVerification"
    }
}
```

Adding the Checkstyle configuration

In order to add the Checkstyle configuration, we need to define the rules against which the code is checked. We can do this by specifying the `config/checkstyle/checkstyle.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
    "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
    "http://www.puppcrawl.com/dtds/configuration_1_2.dtd">

<module name="Checker">
    <module name="TreeWalker">
        <module name="JavadocType">
            <property name="scope" value="public"/>
        </module>
    </module>
</module>
```

Adding a static code analysis stage

We can add a `Static code analysis` stage to the pipeline:

```
stage("Static code analysis") {  
    steps {  
        sh "./gradlew checkstyleMain"  
    }  
}
```

Now, if anyone commits a file with a public class without Javadoc, the build will fail.

Publishing static code analysis reports

Very similar to JaCoCo, we can add the Checkstyle report to Jenkins:

```
publishHTML (target: [  
    reportDir: 'build/reports/checkstyle/',  
    reportFiles: 'main.html',  
    reportName: "Checkstyle Report"  
])
```

It generates a link to the Checkstyle report.

We have added the static code analysis stage that can help in finding bugs and in standardizing the code style inside the team or organization.