Jenkins is an open-source CI/CD WebServer, developed in Java.
Jenkins is highly extensible since many plugins can be installed for our projects' specificities (ie: Git integration).

In this exercise, we will install and configure CI/CD pipelines using Jenkins and its ecosystem.

In order to keep your OS clean, you can use a Docker image.

I - Jenkins Installation and configuration

A - Installing Jenkins

To install Jenkins you can refer to the following link : https://www.jenkins.io/download/
There exist multiple solutions to deploy Jenkins:

- As a web archive (war), if we had a running Java Web Server it could have been an option, but we don't.
- As a docker image.
- Using an installer.

Depending on your context, you may want to choose between a complete installation on your system and a clean, complete Docker environment (https://www.jenkins.io/doc/book/installing/docker/).

B - Preparing Jenkins plugins and settings

1. General setup

Once Jenkins is installed, you should be able to open http://localhost:8080 in a new tab.

You'll be asked to enter a generated password to configure Jenkins for the first time.

We will install the suggested plugins which include Pipelines, Git and Gradle.
Complete the form and create an admin account. Working with accounts is important in CI / CD processes. We should not allow everyone to build or deploy everything.

2. Credentials

We will create credentials in Jenkins to connect to our Github account.
Click on the following links:

- Your username (top right corner)
- Credentials
- User
- Global Credentials
- Add Credentials

3. Useful plugins

In this series, we will create pipelines pulling code from Github and releasing artifacts in Nexus. We will install the following Jenkins plugins:

- Github
- Nexus Platform

You can restart Jenkins using the following URL : http://localhost:8080/safeRestart

II - Integrating Jenkins with Github

We want to trigger CI every time a pull request is opened and every time a merge is performed on the master branch.
There are two ways to achieve this:

- **Solution 1** : We define a cron in the CI server which scans our Github repository every X minutes and runs the CI pipeline if it detects a change.

Here we are creating a lot of overhead (HTTP / CPU / …) and may have a delay between the time the source changed and the time Jenkins fetches the modifications.

- **Solution 2** (using Webhooks) : We define a reactive pipeline by configuring both Github and Jenkins.

The only problem with solution 2 is we need Github to communicate with Jenkins … but we're working on localhost. We'll have to use a tunnel by installing an agent. Many free solutions exist for this. We will use https://smee.io .

A - Create a tunnel

- Using a terminal, install npm
- Install the Smee Client (`npm install --global smee-client`)
- Open https://smee.io and click on "Start a new channel". This will create a unique URL you can use to bind Internet services with your localhost.
- Run the agent with the following command (replace the smee URI with yours). This will expose Jenkins:

smee --url https://smee.io/jUXpBuE5Vb8aVgIo --target http://localhost:8080/github-webhook/

B - Preparing Github

Open the following page: https://github.com/settings/tokens
Click on “Generate new token”. Set the expiration date to 90 days. Give repo and user permissions and save. Note the secret in a file so you don’t lose it.

Go back to your repository and open its settings. Click on “webhooks” and “Add webhook”. The payload URL is your smee.io URL (ie: https://smee.io/jUXpBuE5Vb8aVgIo). Select the application/json content-type and the “Send me everything” option.

Create a file named “Jenkinsfile” in your repository’s root with the following content :

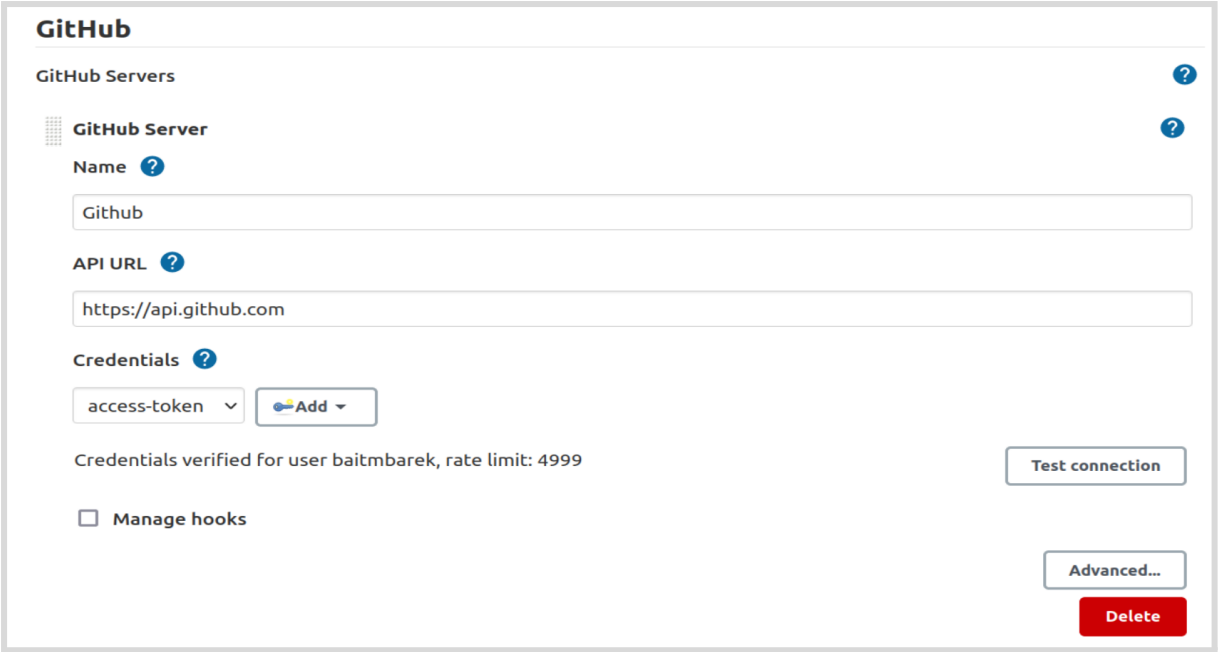
```
pipeline {
 agent any
 stages {
  stage('Script') {
   steps {
    sh 'echo "Hello !"'
   }
  }
 }
}
```

This defines a Jenkins pipeline with only one stage (named script, you can change it) and which runs a shell command to print “Hello !”.

The aim is to test the triggers so it’s enough for now.

C - Jenkins settings

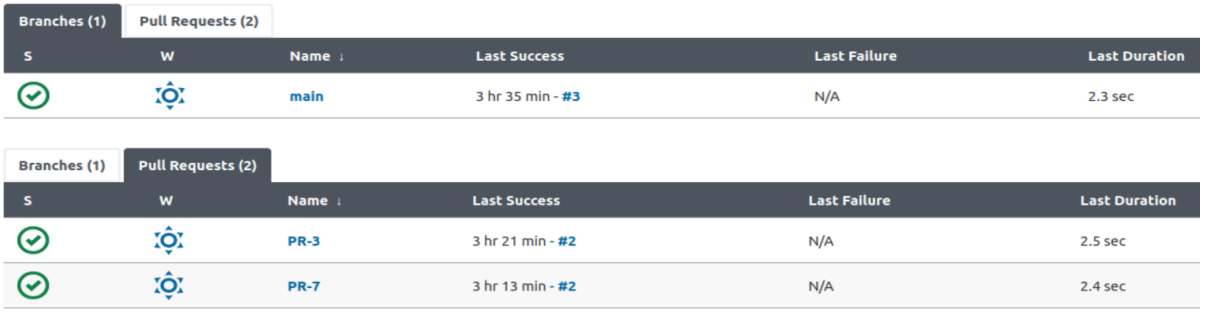
- From the Jenkins Dashboard, click on Manage Jenkins > Manage Credentials. Click on global and then Add credentials.
    - Create a secret text corresponding to your Github access token (saved in file earlier). Give it an ID (ie: access-token)
    - Create a Username with password. The username should be your github account and the password the access token. Give it an ID.
- Go back to the Jenkins Dashboard and click on Manage Jenkins > Configure System. Add a Github server like below:



- Hit the Test connection button and make sure it works.
- Save your configuration.

D - Jenkins pipeline

- From the Jenkins Dashboard, click on “New Item” and create a “Multibranch Pipeline”.
- In the “Branch Sources” section, use your credentials and set your repository URL (Use the HTTPS format, not SSH).
- Save your configuration.
- Jenkins should run a first build for the master branch.
- Open a pull request and check that Jenkins runs the pipeline on the Pull Request branch.
- Merge the pull request and check that Jenkins runs the pipeline on the Master branch.



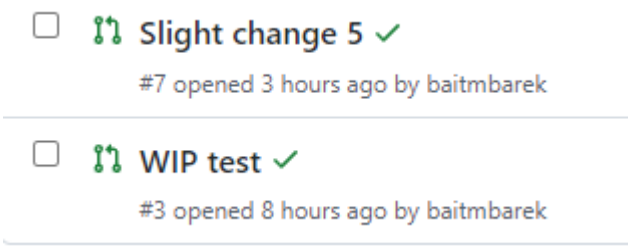
Console Output

```
Push event to branch main
15:33:19 Connecting to https://api.github.com using baitmbarek/******
Obtained Jenkinsfile from 3074e0c7e27e3e511ef9477fbc5db874d7156457
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/mb-app_main
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Checkout SCM)
[Pipeline] checkout
The recommended git tool is: NONE
using credential baitmbarek-apptoken
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/mb-app_main/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/baitmbarek/repo1.git # timeout=10
Fetching without tags
Fetching upstream changes from https://github.com/baitmbarek/repo1.git
 > git --version # timeout=10
 > git --version # 'git version 2.25.1'
using GIT_ASKPASS to set credentials
 > git fetch --no-tags --force --progress -- https://github.com/baitmbarek/repo1.git +refs/heads/main:refs/remotes/origin/main # timeout=10
Checking out Revision 3074e0c7e27e3e511ef9477fbc5db874d7156457 (main)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 3074e0c7e27e3e511ef9477fbc5db874d7156457 # timeout=10
Commit message: "Merge pull request #6 from baitmbarek/dev/wip2"
 > git rev-list --no-walk c6f5b084f10198bc154fc6fa31c0333706ad9acf # timeout=10
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Script)
[Pipeline] sh
+ echo Hello !
Hello !
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline

GitHub has been notified of this commit’s build result

Finished: SUCCESS
```

Did you notice Jenkins sent a confirmation to Github that the build is Successful ? A green mark appears near your successfully built PRs and Branches.



Now, make it fail.

III - Implementing unit tests and building the project

A - Working with Scala

Scala is a programming language widely used in Data Engineering and Microservices development. It runs on the JVM (like Java) and allies both Object Oriented Programming and Functional paradigms.

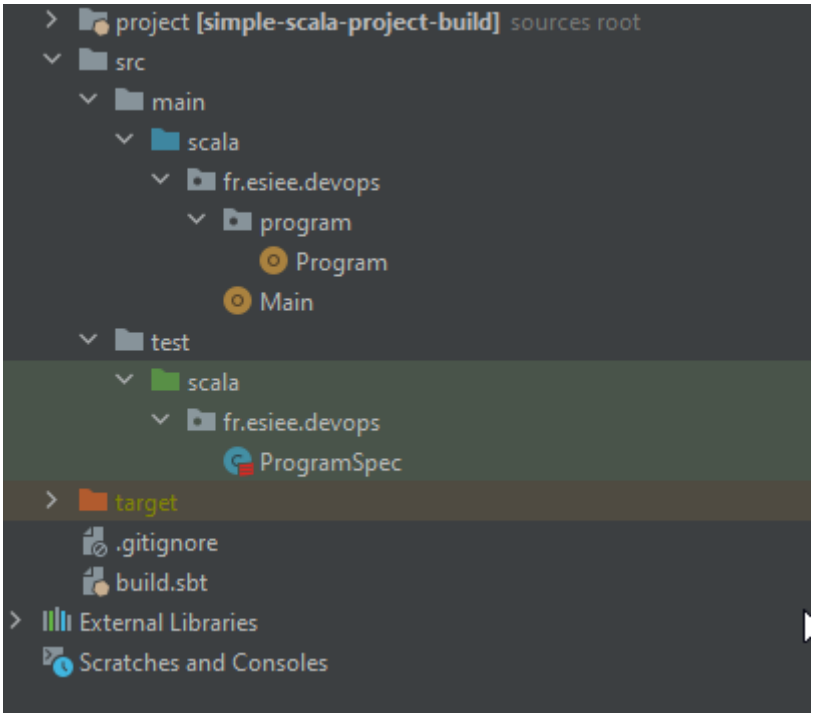
In this session we will be working on the Continuous Integration of a simple Scala project.

Scala is a statically typed language, thus we will need to install some tools before we can use it in our project:

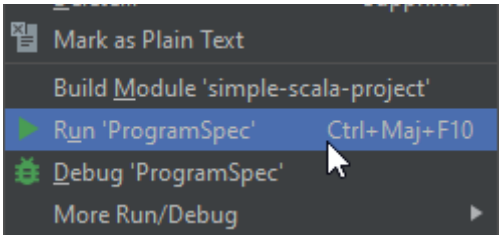
- SBT : Dependency manager and built tool for Scala projects. Instructions can be found here : https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html
- An IDE like IntelliJ with the Scala plugin. You can follow the instructions : https://docs.scala-lang.org/getting-started/intellij-track/getting-started-with-scala-in-intellij.html

B - Testing a Scala Application

- Fork the following repository : https://github.com/baitmbarek/simple-scala-project
- Clone it in your workspace
- Open the project in IntelliJ and make sure it was well imported: You should be able to run the Main object.
- Explore the Program object and the ProgramSpec test.



- You'll need to implement two unit tests (for `Program.lpadZero` and `Program.isPrimeNumber`)
- Once you are done, you should be able to run the ProgramSpec by right-clicking on the file



- You can also run all the tests using SBT from the terminal
    - cd into your project directory
    - Run the command: sbt test

C - Building a new CI pipeline

- If everything's OK, create a Jenkinsfile defining a pipeline with 2 stages:
    - Build Stage => should run : sbt clean compile
    - Test Stage => should run : sbt test
- Add / Commit and Push your changes and the Jenkinsfile **in a new branch**.
- Create a webhook in Github for the forked project.
- Create a new Multibranch Pipeline in Jenkins to build your new repository.
- Create a Pull Request and check that Jenkins triggers the defined stages.

IV - Installing Nexus and publishing artifacts

Congratulations, you have successfully built a project. But what if we want to run it on another machine ? It needs to be packaged somehow in a deliverable (ie: archive / image). You'll also have to either :

- Send it to each server which may want to run it. This requires you to know the full list of potential users, and it will take space in each computer you send it to. What about version management ? Even if this technically works, it's considered a poor solution in particular in a Cloud / Serverless context.
- Store it in a place that can be accessed from outside "on demand". This is what we will do in this section.

A - Setup

We will install Sonatype Nexus which is a secure artifact manager.
Since we already have docker installed, we will run a container for the sonatype/nexus3 image. We will need to map an extra port (ie 9091) in order to expose the docker registry (a bit later).

- First, Nexus requires at least 4GB of free disk space by default. If you don't have much space on your device, consider changing the limit:
    - Access your container as root : docker exec -u 0 -it <containerId> bash
    - Edit the configuration file : /opt/sonatype/nexus/etc/karaf/system.properties
    - Change the following configuration (here I'm using 512MB instead of 4GB): storage.diskCache.diskFreeSpaceLimit=512

    - Save your changes, stop the container and start it again so Nexus reads the property.

- Wait a few seconds for Nexus to start and open it in your browser
- Login as admin
- Create a new Docker hosted repository
    - Add an HTTP connector on the port you mapped earlier.
    - Tick online checkbox
    - Tick “allow anonymous docker pull”
    - Save
- Activate the Docker Bearer Realm (in Security) for authentication
- Open a terminal and type the following (adapt if needed)

```
docker login -u admin localhost:9091
```

- This should succeed.
- Now you can publish images to your private repository:

```
docker pull busybox
docker tag busybox localhost:9091/busybox:latest
docker push localhost:9091/busybox:latest
```

- Browse your docker registry. You should find your image.
- Now, remove busybox from your docker host (both tags)

```
docker rmi busybox localhost:9091/busybox:latest
docker images
```

- Pull busybox from your registry. It should get the image back from Nexus.

Now Nexus is fully setup, we will be able to publish Jars and Docker images from Jenkins.

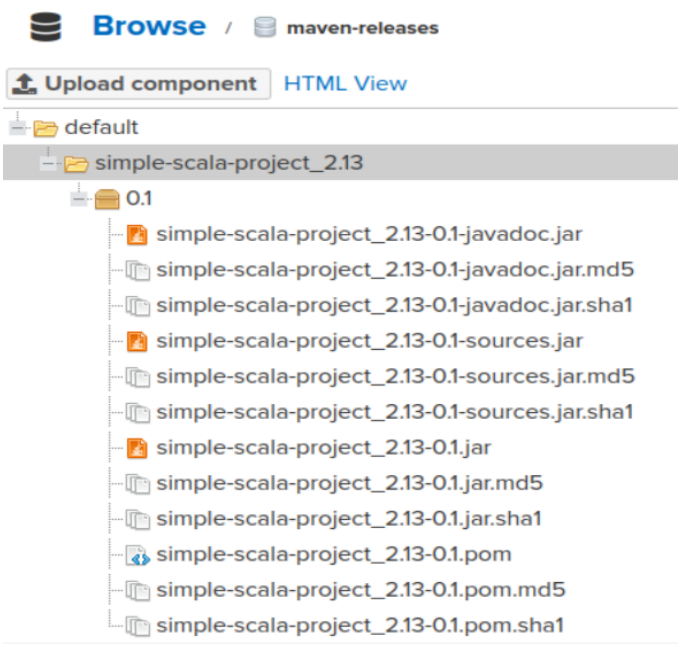
In order to publish Jar files in your Nexus private repository, you’ll need to create the following file : ~/.sbt/.credentials
Its content should be (adapt it) :

```
realm=Nexus Realm
host=localhost
user=jenkins
password=jenkins-password
```

You may need to change SBT default properties to reduce its JVM size:

```
export SBT_OPTS=”-Xmx512M -XX:+UseConcMarkSweepGC
-XX:+CMSClassUnloadingEnabled -Xss2M”
```

Running the “sbt publish” command should send artifacts to Nexus.



B - Releasing projects from Jenkins

- Edit your Jenkins pipeline and add a task which publishes POMs and Jars to Nexus
- Create a Jenkins pipeline for a Docker project (one of yours, in Github) which builds an image and writes it in the Nexus Docker Registry.

Create a “simple” Pipeline job which clones a public Github project, builds it and publishes the associated artifacts to Nexus. You can refer to this URL: http://localhost:8080/job/pipe/pipeline-syntax/

V - Adding integration tests

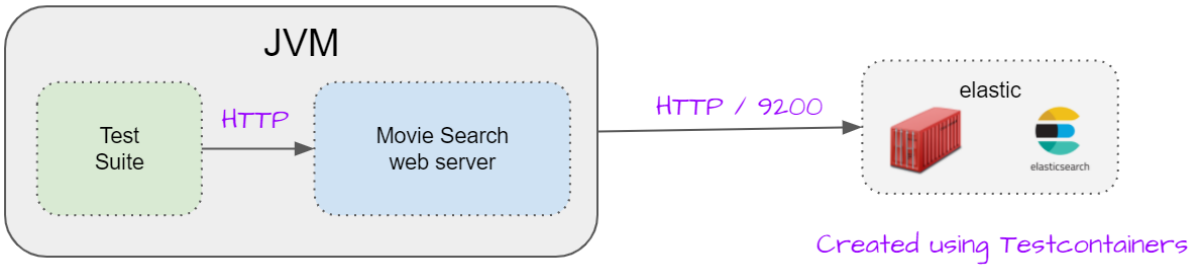
- Fork the following repository if not already done https://github.com/baitmbarek/dataops-movie-search.git
- Clone your repository and import it with IntelliJ.

This project relies on Elasticsearch, so testing it requires interacting with a third-party system. We can either:

1. Mock Elasticsearch, which means we specify how it should answer specific requests
2. Use a dedicated Elasticsearch server : We would have an extra maintenance overhead. This is not efficient at all.
3. Start an embedded or a containerized instance, which we initialize from our test and stop when the tests are done. This solution is strongly recommended.

In this section, we will use the https://testcontainers.org library to manage an Elasticsearch instance.

Integration tests will be performed from an end-user perspective. Once the Elasticsearch container is initialized, the tests will send our service some HTTP requests and expect coherent HTTP responses. We won't call functions or methods explicitly.



Tests are already implemented. All you have to do is define how the container should be instantiated.

- Open the `ElasticContainer` object and implement the *`container`* value.
- Test everything's working on your computer
- Create a Jenkins pipeline which builds this projects, with two additional steps:
  - Run integration tests
  - Publish the docker image to Nexus. This can be done:
    - In two times (using the docker:publishLocal SBT command + docker push)
    - In one time, by configuring SBT.

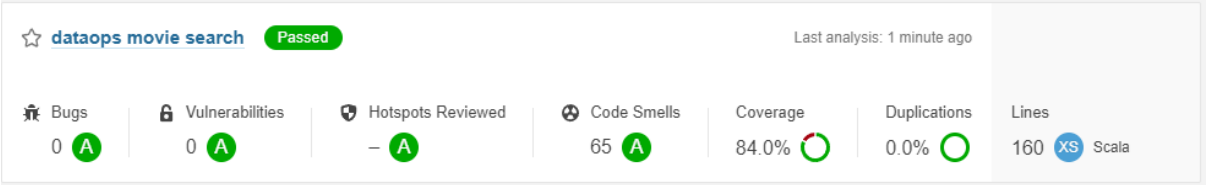
VI - Installing Sonar and publishing reports

Since we're not (yet) building production ready pipelines, we'll follow the "Try out sonarqube" tutorial :

https://docs.sonarqube.org/latest/setup/get-started-2-minutes/

https://docs.sonarqube.org/latest/analysis/jenkins/

- Create a Jenkins account in Sonarqube
- Add a SonarQube step in each of your Scala pipelines. Make sure reports are created in Sonar. Please read and follow the instructions in https://sonar-scala.com/docs/setup/getting-started



- Explore the Code Smells and Coverage scores and try to determine what could have been done to improve these scores.

- Apply some recommendations and improve both Code Smells and Coverage (ignore the Server.scala object).

VII - Recap

Draw a schema summing up what has been done so far.