

TP part 02 - CI/CD



Checkpoint: call us to check your results



Ask yourself: how? why?



Point to document/report

Goals

Good practice

Do not forget to document what you do along the steps.
Create an appropriate file structure, 1 folder per image.

Target application

Complete pipeline workflow for testing and delivering your software application.

We are gonna go through different useful tools to build your application, test it automatically, checking the code quality in the same time.

Useful links :

- <https://docs.travis-ci.com/user/tutorial/>

Setup Travis CI

The first tool we are going to use is Travis CI. Travis is an online service that allows you to build pipelines to test your application. Luckily, the tool is accessible for free as long as your project stays open source. Keep in mind that Travis is not the only one on the market to build integration pipelines. Historically many companies were using [Jenkins](#) (and still a lot continue to do it), it is way less accessible than Travis but much more configurable. You will also hear about [Gitlab CI](#) and [Bitbucket Pipeline](#) during your work life, there are basically as accessible and efficient as Travis, do not hesitate to spend some time learning about those technologies.



Remember those technologies.

Over the years, Travis as well as its concurrents have been improving its implementation to offer always more plug-and-play services. Today, integrating one of these solutions to your project is really easy as child's play. Let's get our hands dirty!

Register to Travis

In the past, you had to sign up to every services you wanted to use, creating a new account, a new password (which was probably the same for each service), filling the same form etc. Technology has improved ever since, and you can now SSO with your Github account to Travis. Even more impressive, it is able to retrieve the information about your Github project (as long as you let them do it), and let you select which project you want to set up. It is a little bit like when Candy Crush asks you to access your Facebook account (except here we do something probably more valuable for our future ;-)).

Well, stop trash talking and [SSO to Travis with your Github account](#).

Select your sample-application-students as a project to be configured and here you are already done with signing up to Travis.

First steps into the CI world

Although it sure is a good thing to be able to link its project to Travis CI, it doesn't make it work by itself (maybe in the future, let's hope).

Most of the CI services use a [yaml](#) file (except Jenkins that uses a... [Groovy](#) file...) to describe the expected steps to be done over the pipeline execution. Go on and create your first `.travis.yml` file into your project's root directory.



Remember the differences between each technology.

Build and test your application

For those who are not familiar with [Maven](#) and [Java](#) project structures, here is the command for building and run your tests :

- mvn clean verify



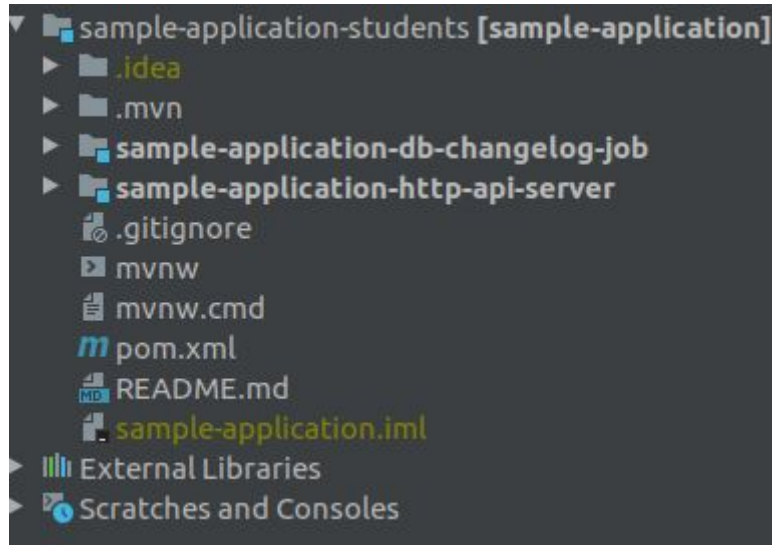
Ok, what is it supposed to do ?

This command will actually clear your previous builds inside your cache (otherwise you can have unexpected behavior because maven did not build again each part of your application), then it will freshly build each module inside your application and finally it will run both [Unit Tests](#) and [Integration Tests](#) (sometime called Component Tests as well).





Unit tests ? Component test ?

Integration tests require a database to verify you correctly inserted or retrieved data from inside. Fortunately for you we've already taken care of this ! But you still need to understand how it works under the hood. Take a look to your application file tree.



There is an application-db-changelog-job which is used to create the schemes inside your database. This service is handled by liquibase that will also take care of version changes in case of new tables creation.

  A db changelog job ?

“Still, how can I run a command that directly launches a docker database in which the db-changelog creates the tables and how do I link it to my application?”

Let's take a look to the pom.xml that is inside the application-http-api-server, you will find some very helpful dependencies for your testing.

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.playtika.testcontainers</groupId>
  <artifactId>embedded-postgresql</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <scope>test</scope>
</dependency>

```

As you can see, there is an “embedded-postgresql” dependency inside the pom. This dependency actually uses the testcontainers.

What are testcontainers?

They simply are java libraries that allow you to run a bunch of docker containers while testing. Here we use the postgresql container to attach to our application while testing. If run the command “mvn clean verify” you’ll be able to see the following:

```

2020-01-27 14:22:48.473 INFO 24545 --- [main] org.testcontainers.DockerClientFactory : Docker host IP address is localhost
2020-01-27 14:22:48.634 INFO 24545 --- [main] org.testcontainers.DockerClientFactory : Connected to docker:
  Server Version: 19.03.5
  API Version: 1.40
  Operating System: Ubuntu 18.04.3 LTS
  Total Memory: 15884 MB
2020-01-27 14:22:49.506 INFO 24545 --- [main] org.testcontainers.DockerClientFactory : Ryuk started - will monitor and terminate Testcontainers containers on JVM exit
  I Checking the system...
  ✓ Docker version should be at least 1.6.0
  ✓ Docker environment should have more than 200 free disk space
2020-01-27 14:22:50.085 INFO 24545 --- [main] [postgres:12.0] : Creating container for image: postgres:12.0
2020-01-27 14:22:50.116 INFO 24545 --- [main] [postgres:12.0] : Starting container with ID: f43bc8dfdf4f0483489d19abe6032cb1069a6d27d72d6329749dedf701c7fc1f
2020-01-27 14:22:50.494 INFO 24545 --- [main] [postgres:12.0] : Container postgres:12.0 is starting: f43bc8dfdf4f0483489d19abe6032cb1069a6d27d72d6329749dedf701c7fc1f
2020-01-27 14:22:52.841 INFO 24545 --- [main] [postgres:12.0] : Container postgres:12.0 started

```

As you can see, a docker container has been launched while your tests were running, pretty convenient, isn’t it? Then you’ll see your db-changelog-job running to create the tables:

```

: SELECT COUNT(*) FROM public.databasechangelock
: CREATE TABLE public.databasechangelock (ID INTEGER NOT NULL, LOCKED BOOLEAN NOT NULL, LOCKGRANTED TIMESTAMP WITHOUT TIME ZONE NOT NULL)
: SELECT COUNT(*) FROM public.databasechangelock
: DELETE FROM public.databasechangelock
: INSERT INTO public.databasechangelock (ID, LOCKED) VALUES (1, FALSE)
: SELECT LOCKED FROM public.databasechangelock WHERE ID=1
: Successfully acquired change log lock
: Creating database history table with name: public.databasechangelog
: CREATE TABLE public.databasechangelog (ID VARCHAR(255) NOT NULL, AUTHOR VARCHAR(255) NOT NULL, FILENAME VARCHAR(255) NOT NULL, LIQUIBASE VARCHAR(255), TAG VARCHAR(255), LIQUIBASE VARCHAR(20), CONTEXTS VARCHAR(255), LABELS VARCHAR(255), DEPLOYMENT_ID VARCHAR(10))
: SELECT COUNT(*) FROM public.databasechangelog
: Reading from public.databasechangelog
: SELECT * FROM public.databasechangelog ORDER BY DATEEXECUTED ASC, ORDEREXECUTED ASC

```

Finally, you'll see your tests results.

Now, it is up to you ! Create your first CI, asking to build and test your application every time someone commits and pushes code on the repository.



Through the Travis CI web console, you will be able to see the logs of your tests. If it passes, the project will be marked as PASSED.

takima-training DEFAULT BRANCH LAST BUILD
sample-application ↗ master passed ✓ #41 passed

  Working CI.

First steps into the CD world



Here we are going to configure the Continuous Delivery of our project. Therefore, the main goal will be to create and save a docker image containing our application on the Docker Hub every time there is a commit on a develop branch.

  Why do we need this branch ?

Create a develop branch in your GitHub Repository.



As you probably already noticed, you need to login to docker hub to perform any publication. However, you don't want to publish your credentials on a public repository (it is not even a good practise to do it on a private repository). Fortunately, Travis allows you to create secured environment variables.

Add your docker hub credentials to the environment variables in Travis CI (and let them secured).

  Secured variables, why ?


Now that you added them, you can freely declare them and use them inside your Travis CI pipeline.

Create dockerfiles to build your applications (db-changelog-job and http-api-server).


  Why do we need this ?

Build your docker images inside your Travis pipeline when there is a commit on the develop branch.

Publish your docker images when there is a commit on the develop branch. (be careful, you need to create the repositories on Docker Hub before publishing them)

 For what purpose ?

Now you should be able to find your docker images on your docker repository.

 Docker images pushed to your repository.

Setup Quality Gate

What is quality about ?

Quality is here to make sure your code will be maintainable and determine every insecured blocks. It helps you producing better and tested features, and it will also prevent having dirty code pushed inside your master branch.

For this purpose, we are going to use SonarCloud, a cloud solution that makes analysis and reports of your code. This is a useful tool that everyone should use in order to learn java best practises.

Register to SonarCloud

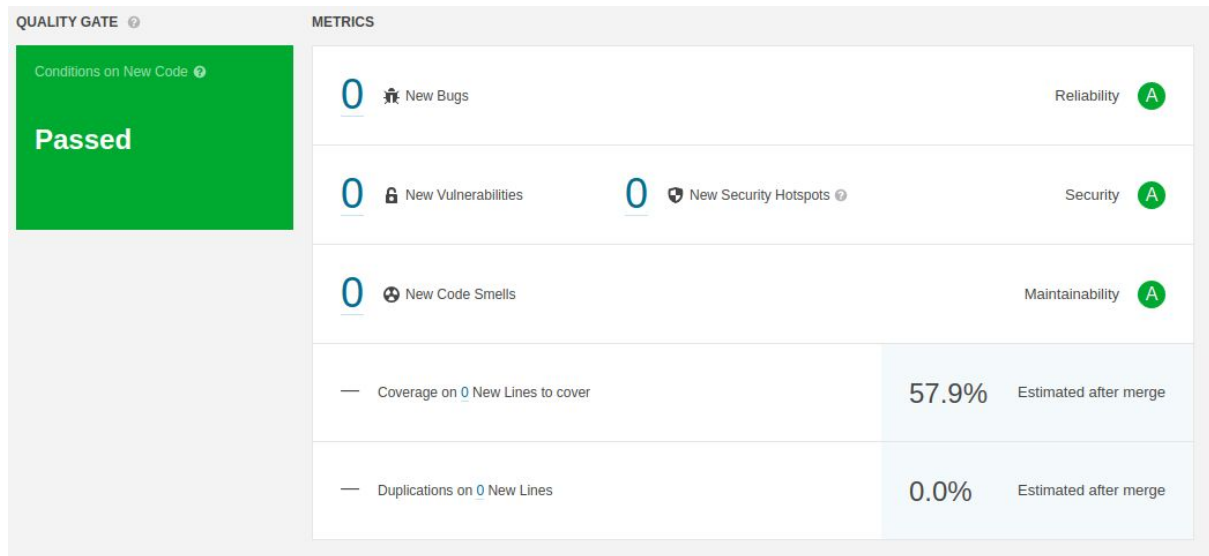
Create your free-tier account on <https://sonarcloud.io/>.

You'll see that you can directly link your GitHub account to your SonarCloud project. Go on and select the repository you want to analyse. Enter your collaboration information etc..

SonarCloud will propose you to setup your Travis CI pipeline from the Travis CLI, but forget about that, there is a much better way to save the SonarCloud provided and provide it into your .travis.yml.

Setup your pipeline to use SonarCloud analysis while testing.

If you did your configuration correctly, you should be able to see the SonarCloud analysis report online :



  Working quality gate.

Well done buddies, you've created your very first Quality Gate ! Yay !


Going further

Till now, I suppose your Travis CI script executes everything inside one unique step. The problem is, if anything fails somewhere, the rest of the script will still be executed. You don't want your failed build to be deployed into your Docker Hub Repository.

Moreover, it is difficult to select scripts to be executed on a specific branch. Deploy your environment when you publish on a feature branch doesn't really make sense.

That's why you can create jobs inside your [Travis pipeline](#). Jobs allow you to split your scripts and select which branch should run what.

Now split your pipeline to separate the test and the builds from the deployment of your images on Docker Hub.

  Splitted pipeline into jobs.

Well done, you went till the end of this practical application, now move on to the deployment section with Ansible !