**Compte rendu TP3: Cloud DevOps**

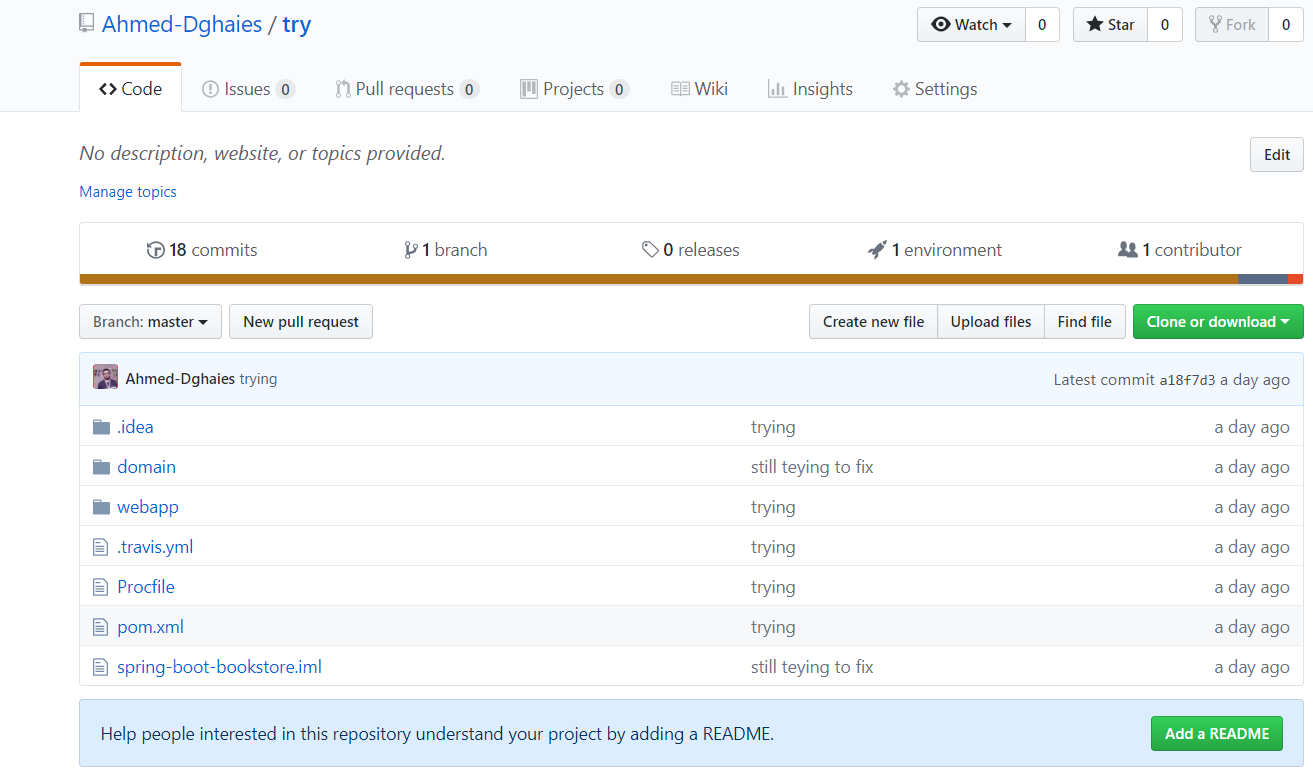
Réalisé par :  
Dghaies Ahmed  
Samet Mouhamed Amine

**Activity 1 : Travis CI**

In this activity we are going to learn how we can use **Travis** **CI** for Continuous Integration and Continuous Deployment (CI/CD) of a **SpringBoot** application. We will learn how to run maven build goals, perform test coverage validation using JaCoCo plugin, Code Quality checks using **SonarCloud** and finally deploy it to **Heroku**.

**Step 1: Create SpringBoot project**

We can create a simple SpringBoot project using <https://start.spring.io> or using intelIJ after that we should push it to a github repository



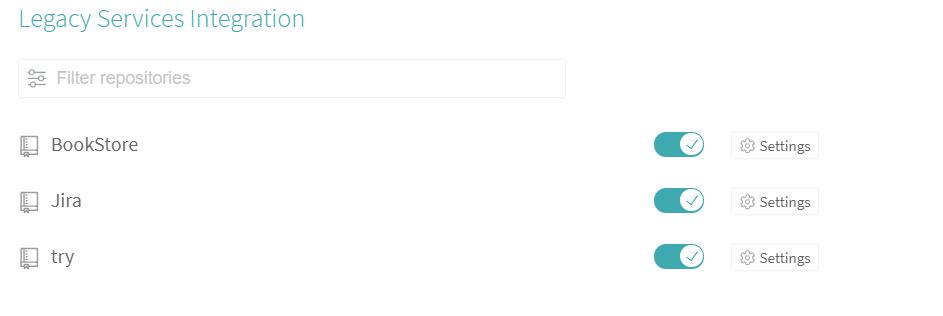
## Step 2: Create .travis.yml file

For Travis CI to work we need to add .travis.yml file to our project root folder. For it to work it will just need a minimal configuration, just language and jdk.

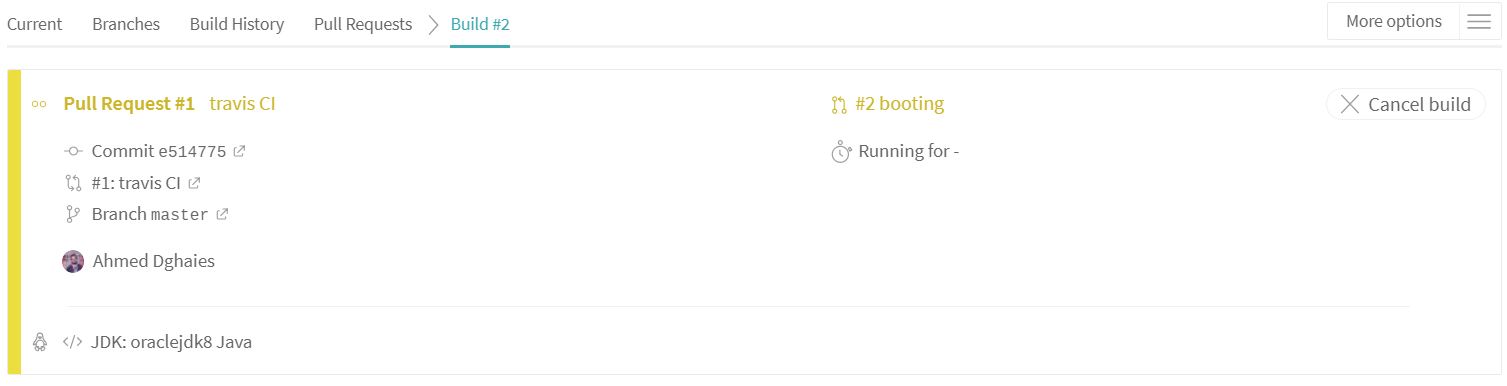


## Step 3: Enable Travis-CI for GitHub repository

We need to go to <https://travis-ci.org/> and signin with github

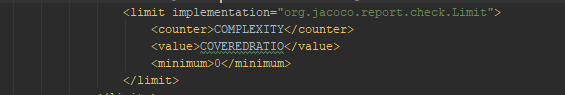


And as you can see you should enable each repository on its own. After enabling a repository you can go to more options > Trigger build and see the build running



## Step 4: Add JaCoCo Code Coverage check

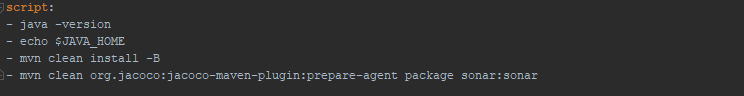
Here we should add a JaCoCo plugin for coverage check into our pom.xml file with options like the desired percentage, for this report purpose I decided to set the percentage to 0 (it should be set to 60%)



## Step 5: Run Unit and Integration Tests

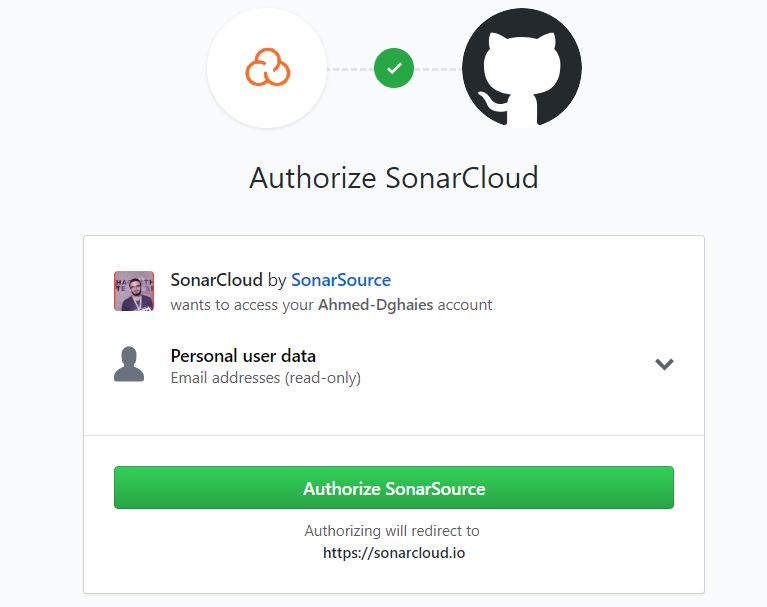
Now by default Travis runs mvn test –B which only run Unit tests, we want to run Unit tests and Integration tests separately by using maven-failsafe-plugin.

Now we are going to use **script** block to specify our custom maven goal to run instead of default goal. For me I set four goals

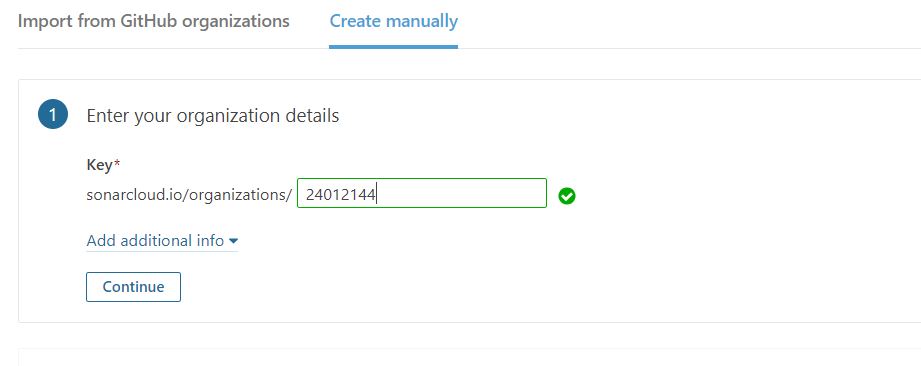


## Step 6: SonarQube code quality checks using SonarCloud

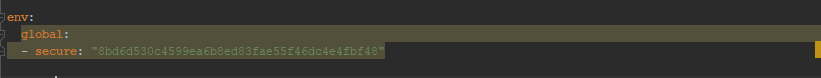
[SonarCloud](https://sonarcloud.io/) , which built on **SonarQube**, offers free code quality checks for Open Source projects.



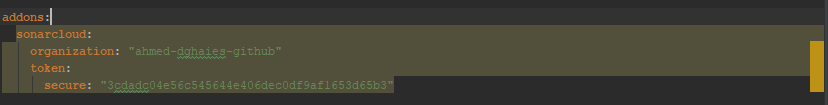
We also need to create an organization with a unique key for our application



We can add all the secrets as global environment variables as follows:



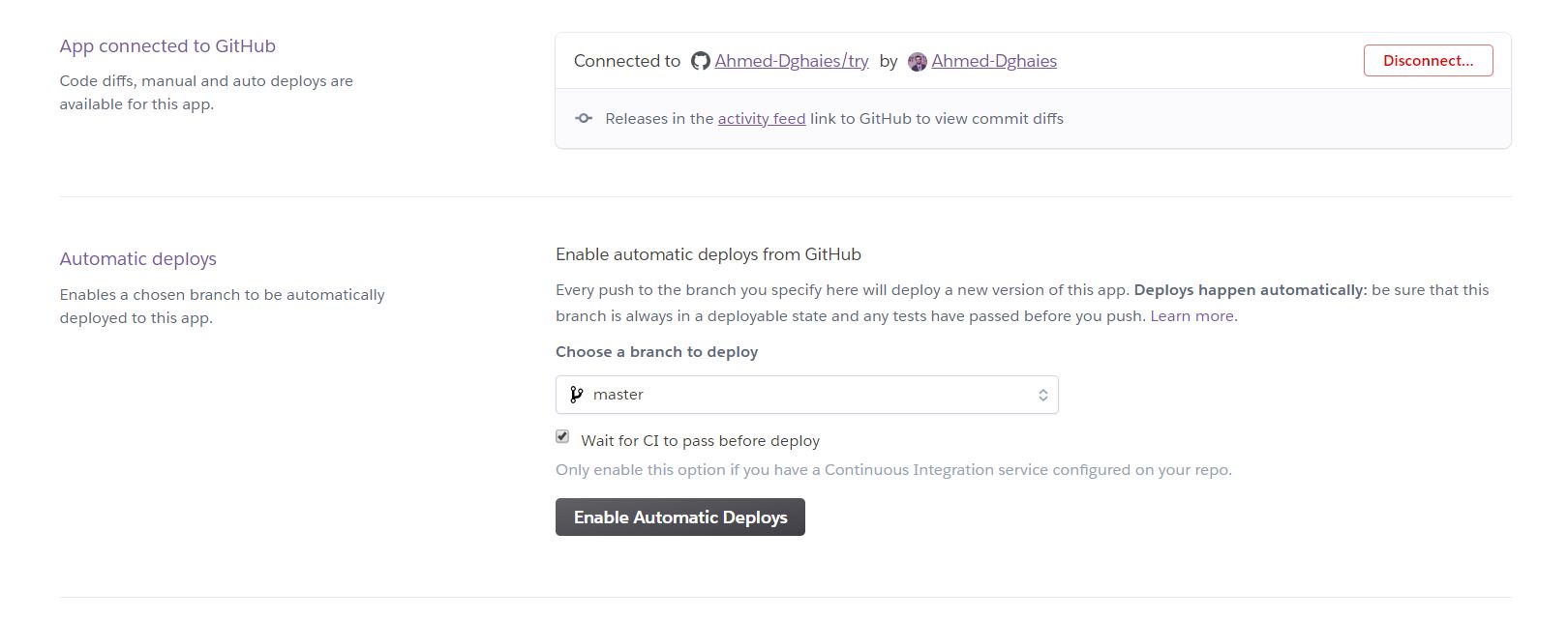
Finally, we should add SonarCloud support as an AddOn:



## Step 8: Deploy to Heroku

Travis CI provides options to deploy on a wide range of platforms including Heroku, OpenShift, AWS, Azure etc. Travis CI can automatically deploy your Heroku application after a successful build.

We are going to deploy our SpringBoot application on Heroku using Travis <https://docs.travis-ci.com/user/deployment/heroku/>. Before deploying our application to Heroku first we need to login to <https://www.heroku.com/> and create an application from Dashboar



Now create **Procfile** in root folder of the project as follows:

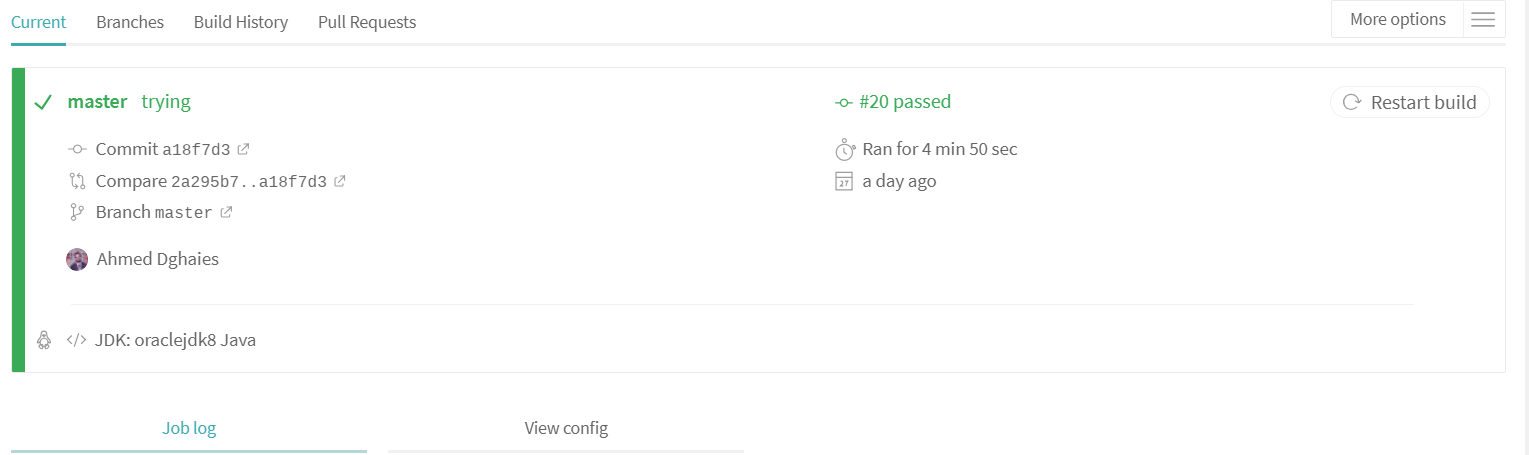


And not to forget to add deploy to .travis.yml

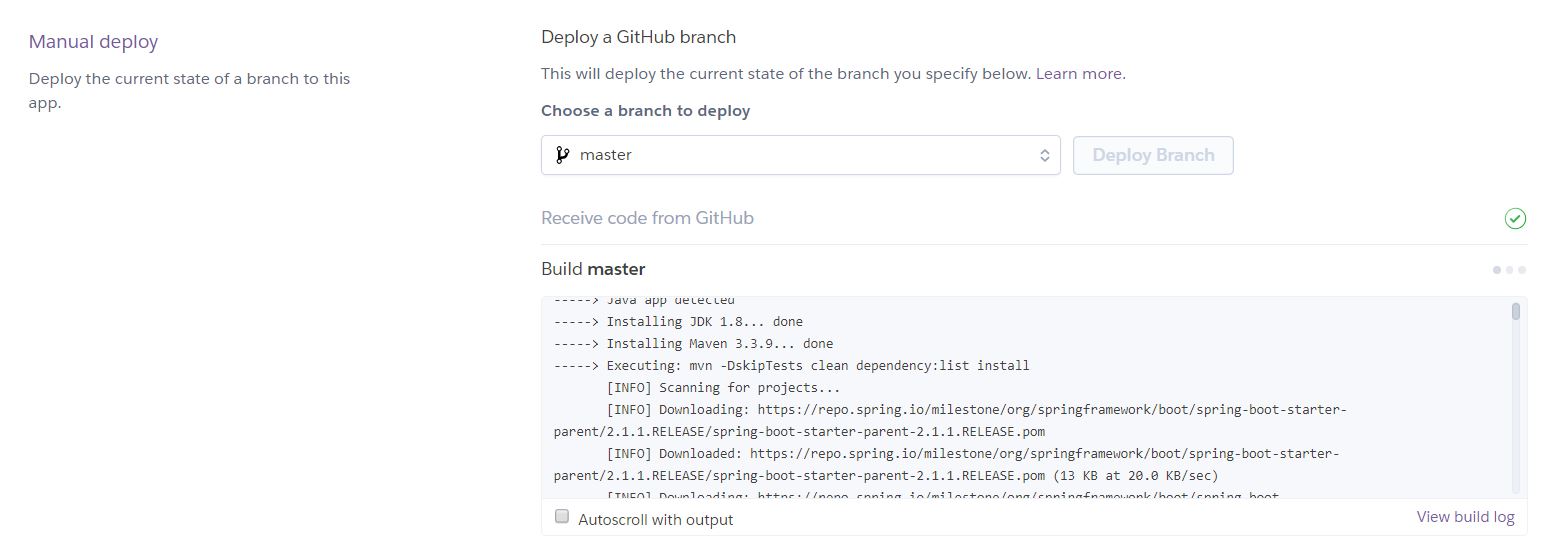


## Step 9: Launch

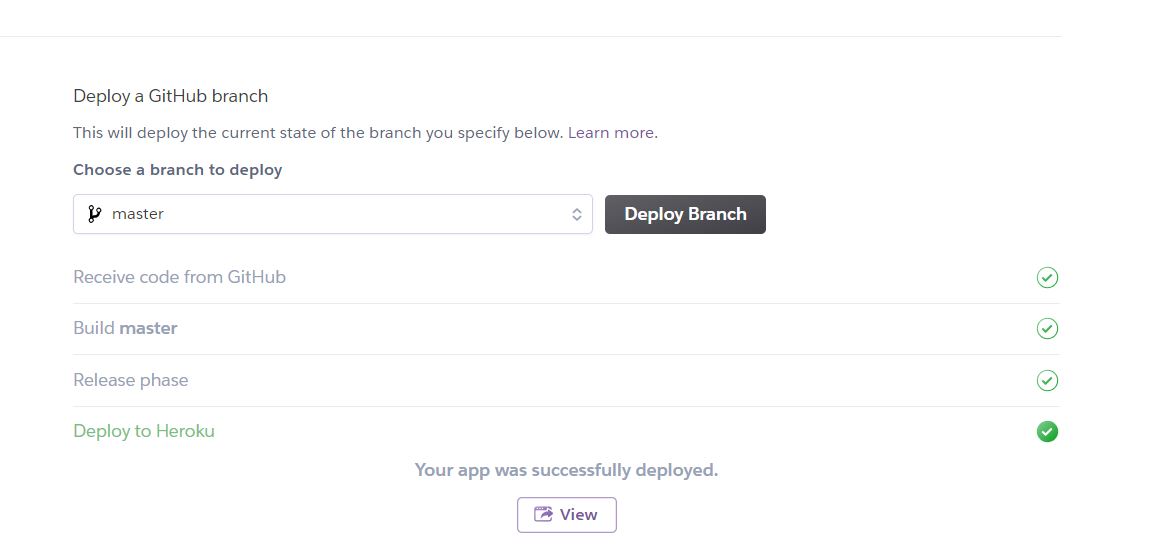
Once the build is successful we can see it in the Travis dashboard



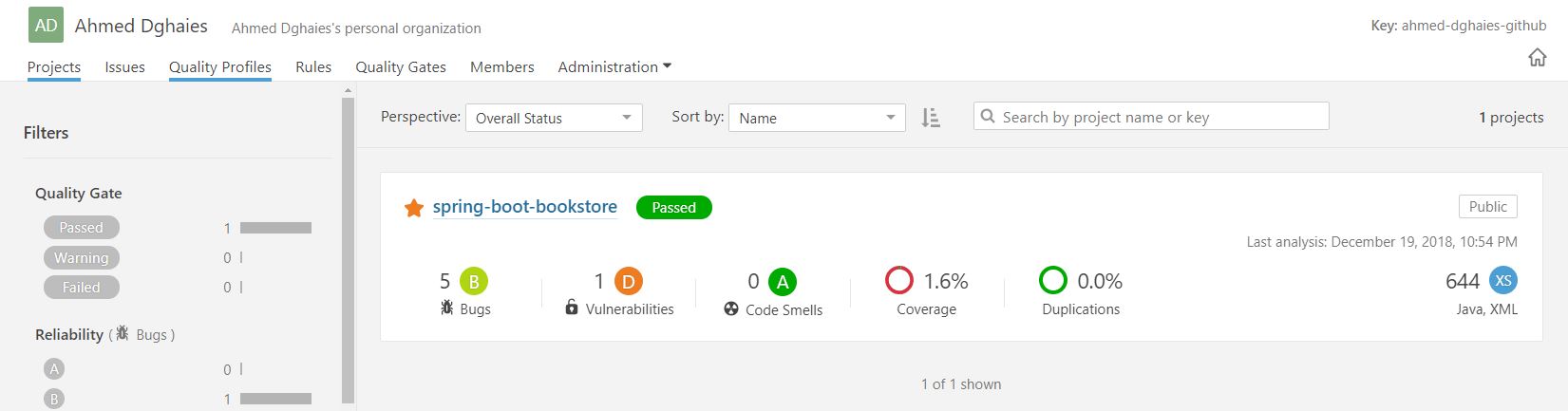
Here we can see Heroky deploying



And the result

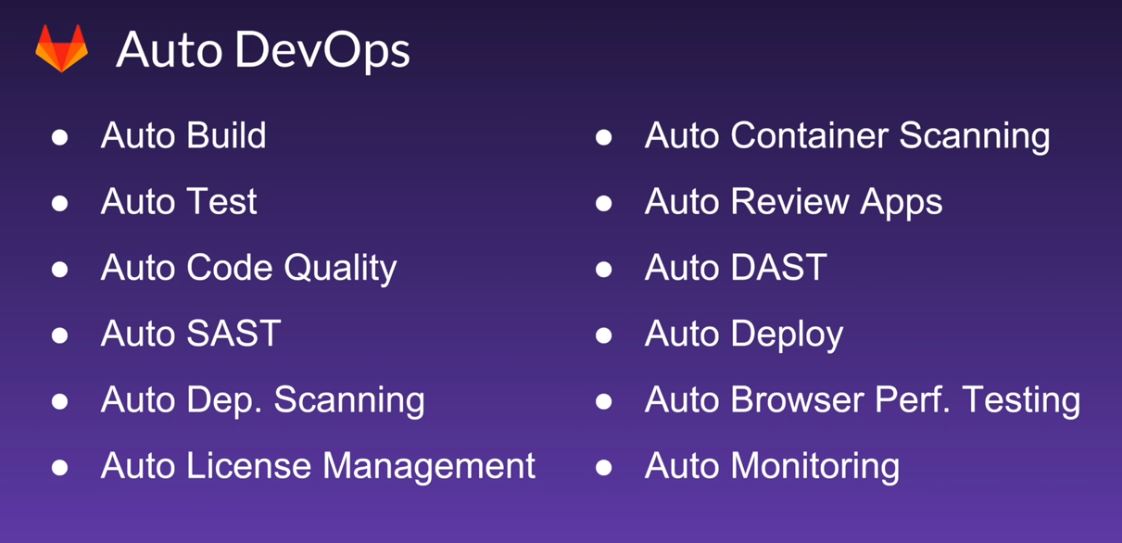


And sonarCloud results all automated

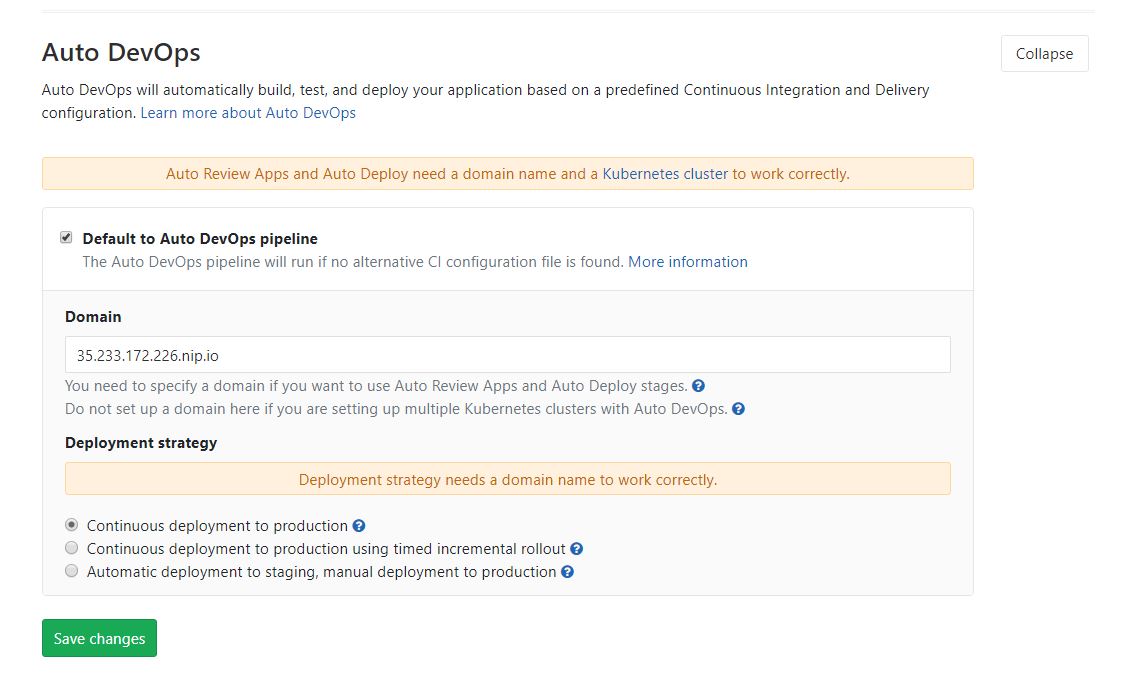


**Activity 2 : Azure DevOps**

Auto DevOps provides pre-defined CI/CD configuration which allows you to automatically detect, build, test, deploy, and monitor your applications. Leveraging CI/CD best practices and tools, Auto DevOps aims to simplify the setup and execution of a mature & modern software development lifecycle



Let’s start by enabling Auto DevOps for our repository

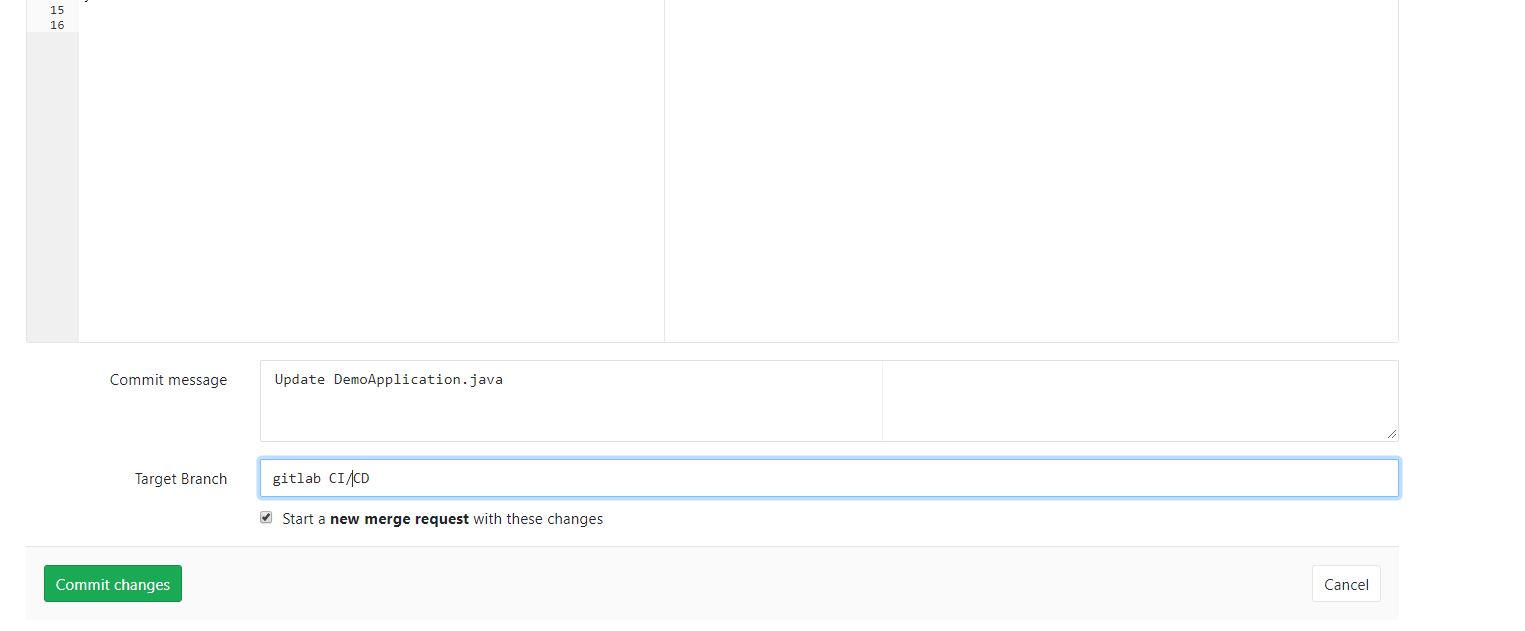


You can change the deployment strategy used by Auto DevOps by going to your project’s **Settings > CI/CD > Auto DevOps**.

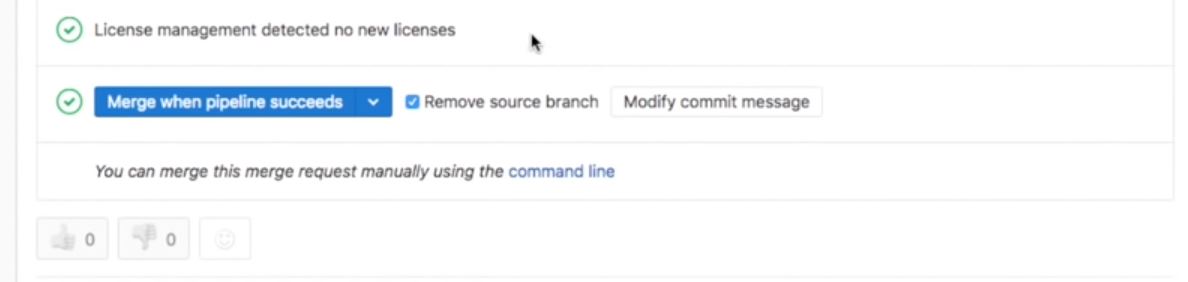
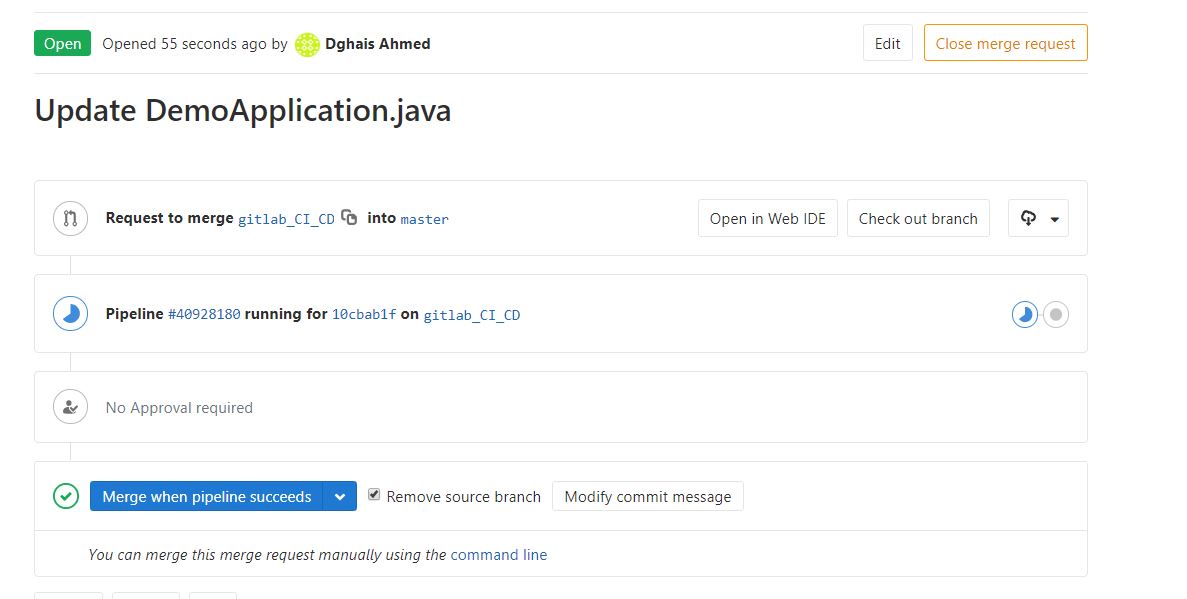
The available options are:

* **Continuous deployment to production**: Enables [Auto Deploy](https://docs.gitlab.com/ee/topics/autodevops/#auto-deploy) with master branch directly deployed to production.
* **Continuous deployment to production using timed incremental rollout**: Sets the [INCREMENTAL\_ROLLOUT\_MODE](https://docs.gitlab.com/ee/topics/autodevops/#timed-incremental-rollout-to-production) variable to timed, and production deployment will be executed with a 5 minute delay between each increment in rollout.
* **Automatic deployment to staging, manual deployment to production**: Sets the [STAGING\_ENABLED](https://docs.gitlab.com/ee/topics/autodevops/#deploy-policy-for-staging-and-production-environments) and [INCREMENTAL\_ROLLOUT\_MODE](https://docs.gitlab.com/ee/topics/autodevops/#incremental-rollout-to-production) variables to 1 and manual. This means:
  + master branch is directly deployed to staging.
  + Manual actions are provided for incremental rollout to production.

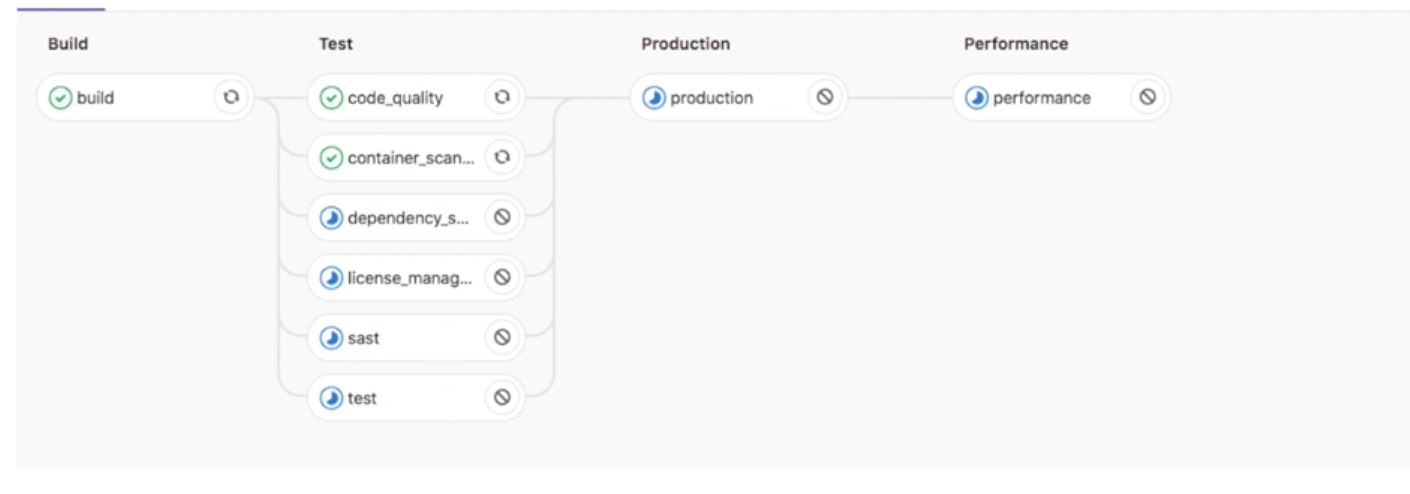
In order to understand Auto DevOps let’s put it in action, we made a little change, pushed it into a new branch and then merged



Now we can see that Auto DevOps starts a pipeline on its own



After merging we can move to production and then we can see a little change



Once your application is deployed, Auto Monitoring makes it possible to monitor your application’s server and response metrics right out of the box. Auto Monitoring uses Prometheus to get system metrics such as CPU and memory usage directly from Kubernetes, and response metrics such as HTTP error rates, latency, and throughput from the NGINX server.

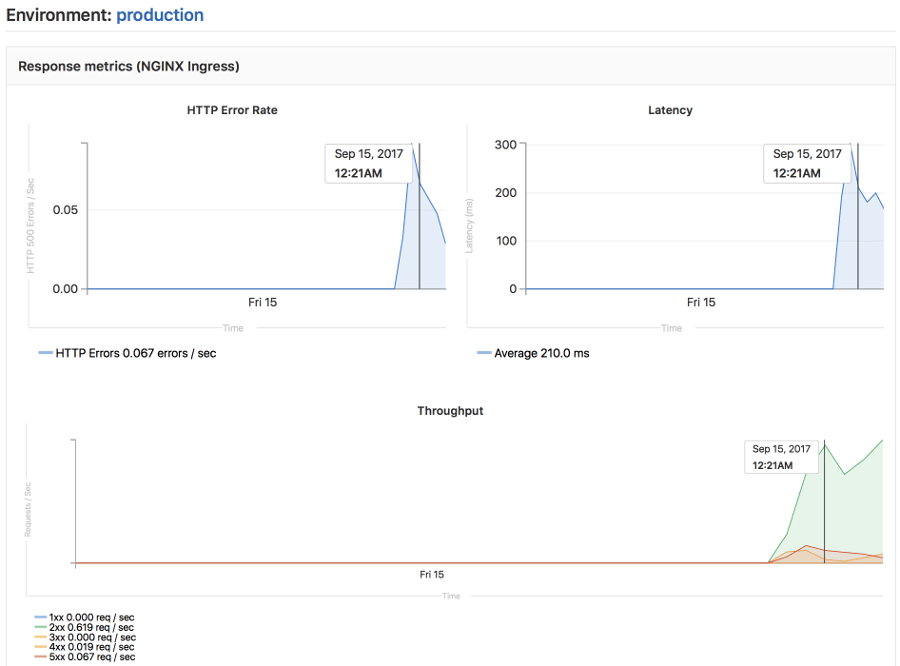
The metrics include:

* **Response Metrics:** latency, throughput, error rate
* **System Metrics:** CPU utilization, memory utilization

In order to make use of monitoring you need to:

1. Deploy Prometheus into your Kubernetes cluster
2. If you would like response metrics, ensure you are running at least version 0.9.0 of NGINX Ingress and enable Prometheus metrics.
3. Finally, annotate the NGINX Ingress deployment to be scraped by Prometheus using prometheus.io/scrape: "true" and prometheus.io/port: "10254".

To view the metrics, open the Monitoring dashboard for a deployed environment.



If we want to change the deployment from automated to manual we need to check this option in the deployment strategy:

