MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DISTRIBUTED SYSTEMS

CI/CD Tutorial and Deployment on cloud (Heroku Cloud)

Ioan Salomie

Tudor Cioara
Claudia Antal

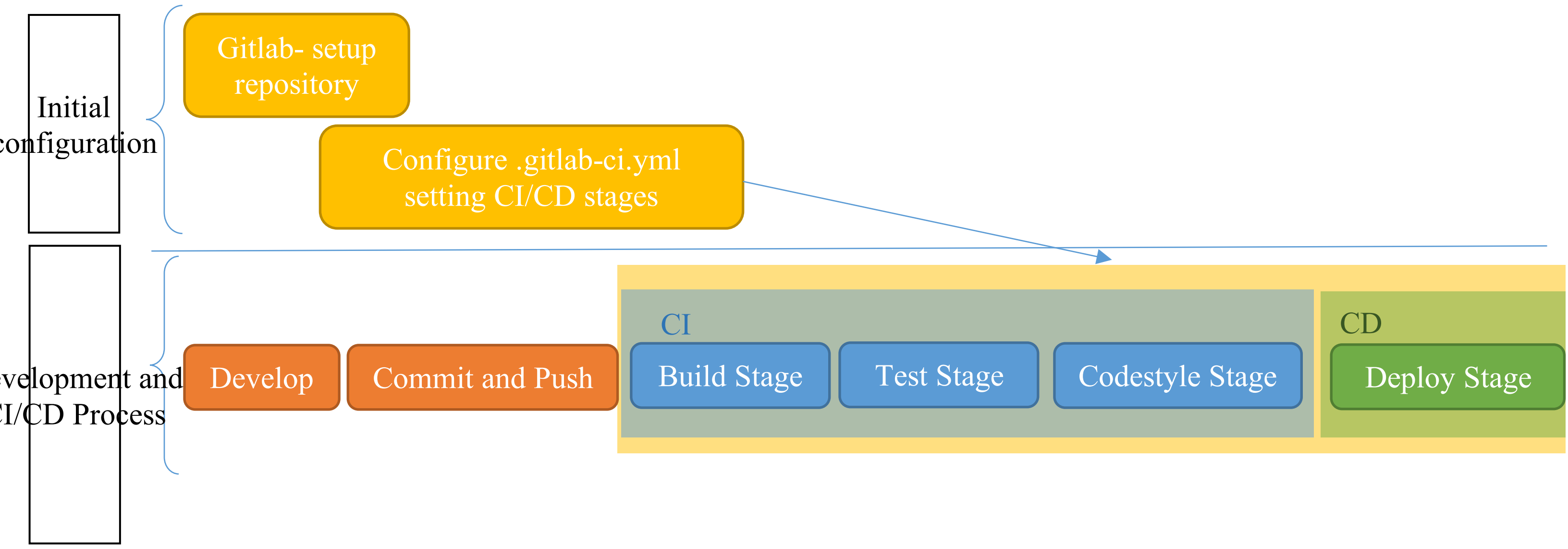
Marcel Antal

2021-2022

Contents

1. Overview

From this tutorial you will learn how to configure the CI/CD pipeline in Gitlab for a spring-boot application. You can use the source code provided in [1] and [2] and setup your own repository on Gitlab and following the instructions and the exercises from *"Test your solution"*. By the end of the laboratory you should have your own backend and frontend application configured to run both the CI and CD pipeline.



For setting your Gitlab repositories:

1. First setup a Gitlab Group by going to: *Groups →Your Groups → New Group* and create your *private* group for the DS project as:
   DS2020_GroupNumber_LastName_FirstName
   (e.g. DS2020_30441_Popescu_Ioan)
2. Give access to your group for the DS lab assistants. On your Group page go to:
   Members → Invite Member →
   and offer **Maintainer** rights for the user*: utcn.dsrl@gmail.com*
3. Inside the group, you can create your own projects for different applications of the DS lab. Make sure you keep the naming conventions for the projects as well, considering one of the following formats:
   - *DS2020_GroupNumber_LastName_FirstName_AssigNumber*
   - *DS2020_GroupNumber_LastName_FirstName_AssigNumber_Backend*
   - *DS2020_GroupNumber_LastName_FirstName_AssigNumber_Frontend*

2. Continuous Integration (CI) - Backend

Continuous Integration (CI) refers to a pipeline of steps that are applied whenever your code is pushed on the code repository. It aims to validate that the code you developed does not affect the previously developed features and that the integration between your newly developed code and the previous code is done correctly. Specifically, this can be done in 3 steps:

1. Verify that the project builds correctly
2. Verify that the tests run and are successful
3. Verify that there are no major code style issues

In the following sections each of these verification steps will be addressed.

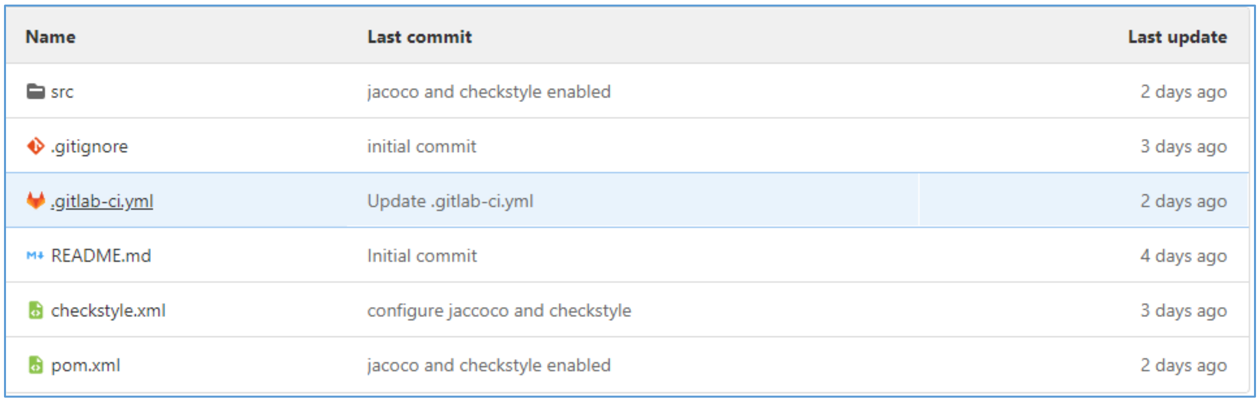
**) Additional information and the source code for this tutorial can be find at [1]*

Exercise: You can download the source code from [1] and upload it in your own Gitlab repository or push your own Spring Boot source code online.

2.1. Setting the gitlab-ci file

In order to trigger the pipeline of continuous integration on Gitlab, it is necessary to add a .gitlab-ci.yml configuration file, where you need to specify all the pipeline steps together with the commands and configurations necessary to successfully run that step.

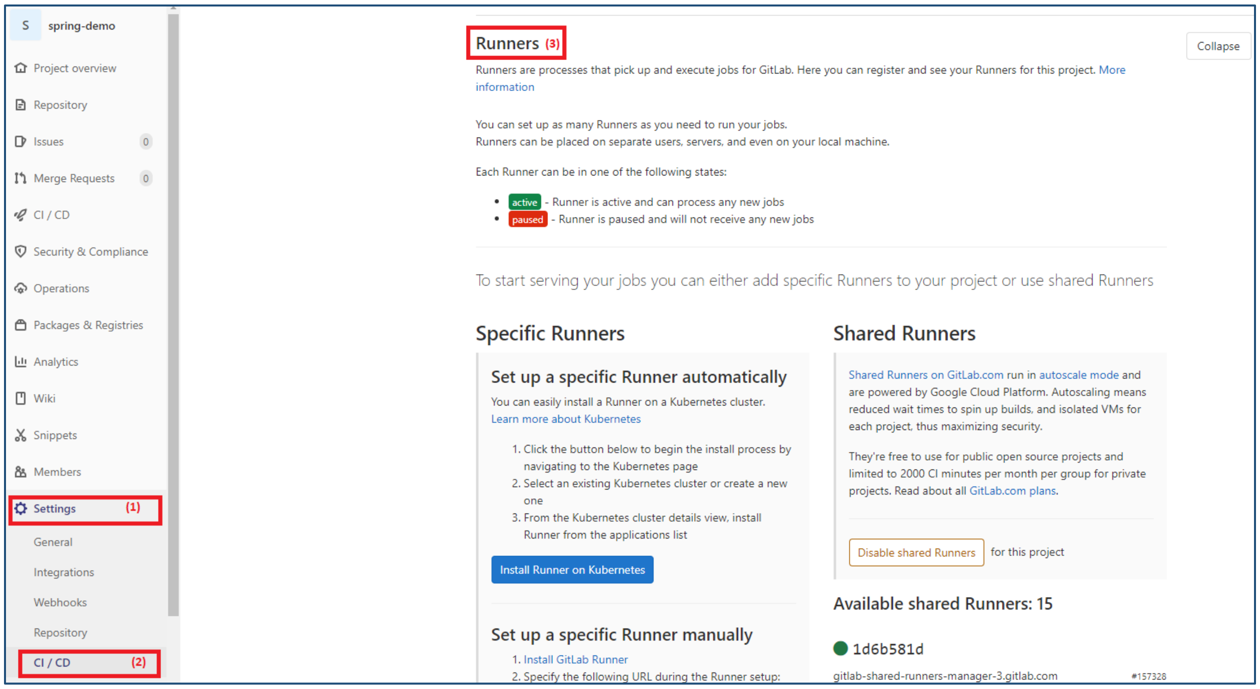
The .gitlab-ci.yml need to be registered in the root of the project as depicted in Figure 1.



Name	Last commit	Last update
src	jacoco and checkstyle enabled	2 days ago
.gitignore	initial commit	3 days ago
.gitlab-ci.yml	Update .gitlab-ci.yml	2 days ago
README.md	Initial commit	4 days ago
checkstyle.xml	configure jaccoco and checkstyle	3 days ago
pom.xml	jacoco and checkstyle enabled	2 days ago

Figure 1 Project structure for setting CI on Gitlab

In order to validate that the CI/CD configuration is possible, make sure that you have the Runners enabled for your project.
For this go to: *Settings* → *CI/CD* → *Runners*



Runners (3)

Runners are processes that pick up and execute jobs for GitLab. Here you can register and see your Runners for this project. More information

You can set up as many Runners as you need to run your jobs.
Runners can be placed on separate users, servers, and even on your local machine.

Each Runner can be in one of the following states:

- active - Runner is active and can process any new jobs
- paused - Runner is paused and will not receive any new jobs

To start serving your jobs you can either add specific Runners to your project or use shared Runners

Specific Runners

Set up a specific Runner automatically

You can easily install a Runner on a Kubernetes cluster. Learn more about Kubernetes

1. Click the button below to begin the install process by navigating to the Kubernetes page
2. Select an existing Kubernetes cluster or create a new one
3. From the Kubernetes cluster details view, install Runner from the applications list

Set up a specific Runner manually

1. Install GitLab Runner
2. Specify the following URL during the Runner setup:

Shared Runners

Shared Runners on GitLab.com run in autoscale mode and are powered by Google Cloud Platform. Autoscaling means reduced wait times to spin up builds, and isolated VMs for each project, thus maximizing security.

They're free to use for public open source projects and limited to 2000 CI minutes per month per group for private projects. Read about all GitLab.com plans.

Disable shared Runners for this project

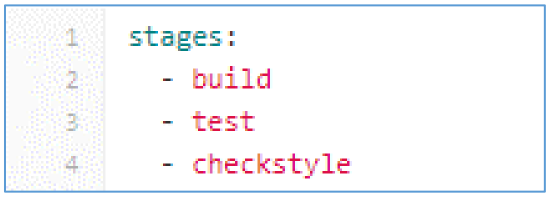
Available shared Runners: 15

● 1d6b581d
gitlab-shared-runners-manager-3.gitlab.com #157328

Figure 2 Check Runners are enabled

By default the runners are enabled, thus you do not have to do anything.

In the .gitlab-ci.yml file we configure the 3 stages (Figure 3) required to be run whenever new code is pushed in the repository:

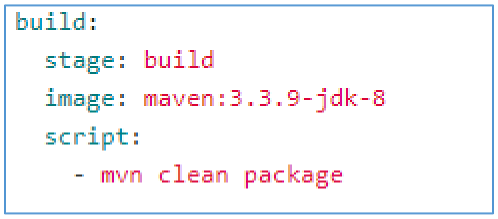


```
stages:
  - build
  - test
  - checkstyle
```

Figure 3 CI Pipeline stages

2.1.1. Setting the Build phase

During the build phase, the build command is issued on the source code that has been updated:



```
build:
  stage: build
  image: maven:3.3.9-jdk-8
  script:
    - mvn clean package
```

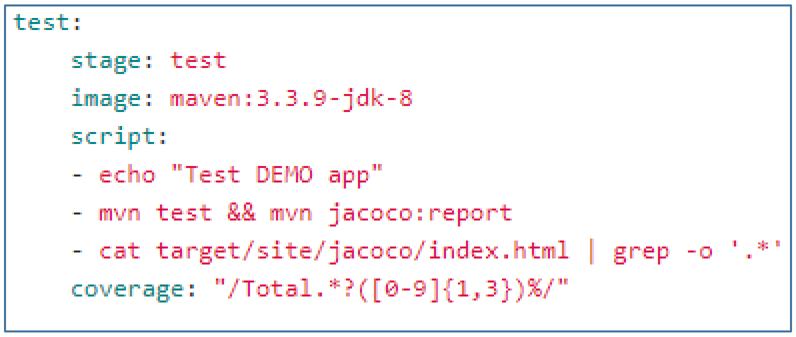
Figure 4 Build Stage

Having a Spring Boot application configured using Maven, in order to build our code we need to specify that this stage needs to be executed on an image preconfigured with Maven.
Furthermore, using the *script* tag, the command needs to be given that will be executed for this stage.

Exercise: You may change the command and specify mvn clean install

2.1.2. Setting the Testing phase

During the test stage, all the tests are expected to be run in order to detect any possible problems that may have raised as a result of updating the code.

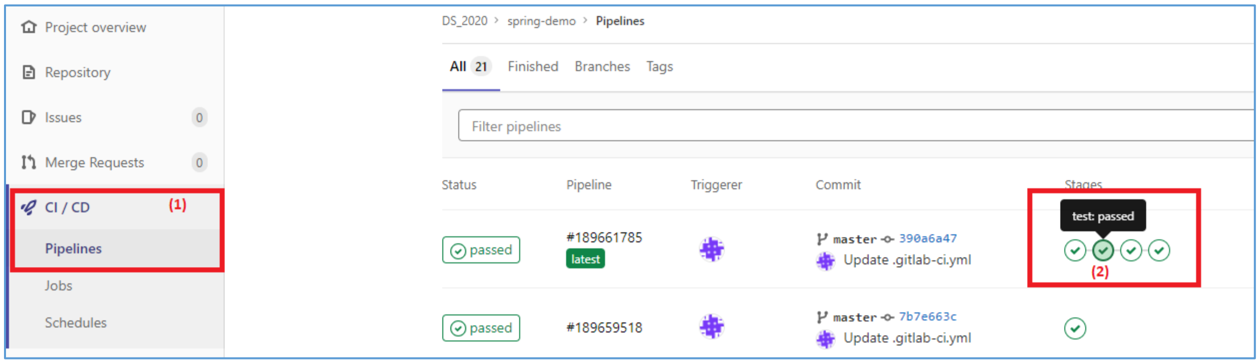


```
test:
    stage: test
    image: maven:3.3.9-jdk-8
    script:
    - echo "Test DEMO app"
    - mvn test && mvn jacoco:report
    - cat target/site/jacoco/index.html | grep -o '.*'
    coverage: "/Total.*?([0-9]{1,3})%/"
```

Figure 5 Test Stage

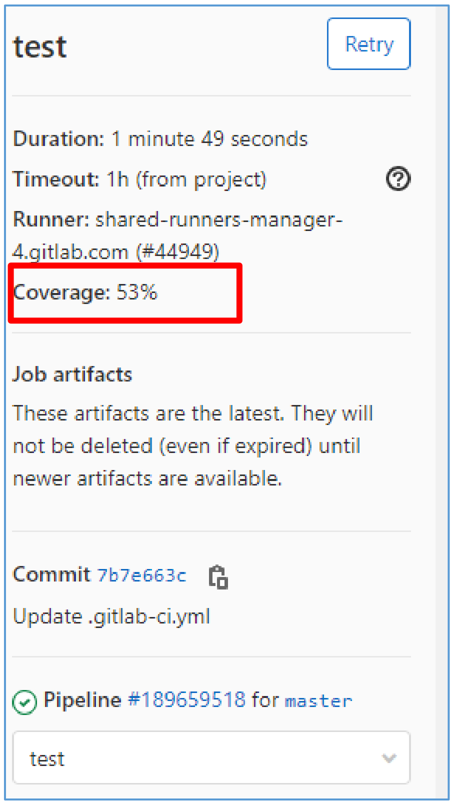
Similarly to the build stage, we start from a maven image, however, the script tag contains the maven command for executing the tests.
Additionally, the jacoco plugin is used (check the project's pom.xml lines 94-151) in order to generate the test reports. This is useful in order to extract useful information about the testing phase, such as the test coverage (the percentage of lines of code covered by tests).
As a result the test coverage is depicted in the test stage details. In order to see the test details:

Select *CI/CD* → *Pipelines* → Select and Click the *Test* phase from the Pipeline → Test Coverage is depicted on the right side of the screen (Figure 7)



Status	Pipeline	Triggerer	Commit
passed	#189661785 latest		master 390a6a47 Update .gitlab-ci.yml
passed	#189659518		master 7b7e663c Update .gitlab-ci.yml

Figure 6 Test Stage Selection



test

Retry

Duration: 1 minute 49 seconds

Timeout: 1h (from project)

Runner: shared-runners-manager-4.gitlab.com (#44949)

Coverage: 53%

Job artifacts

These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.

Commit 7b7e663c

Update .gitlab-ci.yml

Pipeline #189659518 for master

test

Figure 7 Test Stage Report

2.1.3. Setting the Check style phase

During the check style phase (or the linting phase) the code is evaluated in order to detect possible coding style errors or warnings. These errors can be signs of suspicious constructs that can lead to future programming errors.

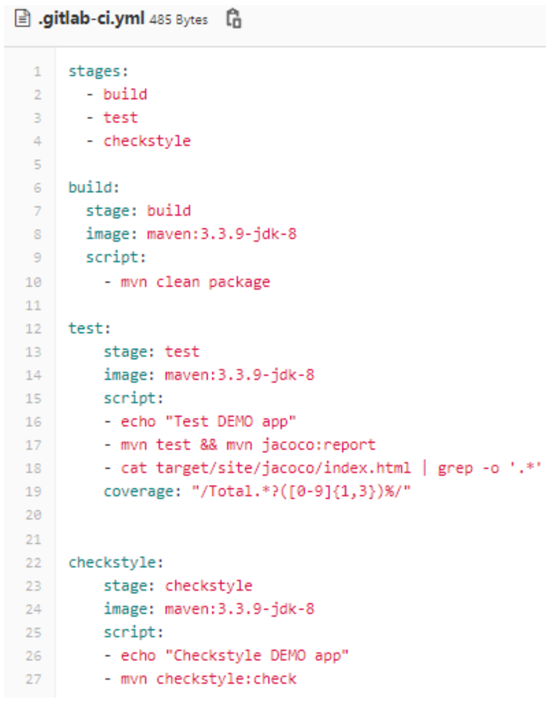
```
checkstyle:
    stage: checkstyle
    image: maven:3.3.9-jdk-8
    script:
    - echo "Checkstyle DEMO app"
    - mvn checkstyle:check
```

Figure 8 Checkstyle Stage

For the current pipeline the checkstyle plugin is used (see the project's pom.xml [1]). The plugin relies on a configuration file (checkstyle.xml – see the project structure [1]) that specifies which coding errors / warning to be investigated when the linter is applied. For official configurations, check the 'Google Java Style' xml at [3].
Whenever the check phase encounters an error in your coding style, the check style phase will fail.

2.2. Test your solution

By this point your .gitlab-ci.yml should look like this:



.gitlab-ci.yml 485 Bytes

```yaml
stages:
  - build
  - test
  - checkstyle

build:
  stage: build
  image: maven:3.3.9-jdk-8
  script:
    - mvn clean package

test:
    stage: test
    image: maven:3.3.9-jdk-8
    script:
    - echo "Test DEMO app"
    - mvn test && mvn jacoco:report
    - cat target/site/jacoco/index.html | grep -o '.*'
    coverage: "/Total.*?([0-9]{1,3})%/"


checkstyle:
    stage: checkstyle
    image: maven:3.3.9-jdk-8
    script:
    - echo "Checkstyle DEMO app"
    - mvn checkstyle:check
```

Figure 9 CI setup

Make a modification on your code, locally and then push it to your Gitlab account.

- Check the CI/CD pipelines and make sure that the pipeline is triggered once your code is loaded on git.
- If the pipeline fails, make sure you make the necessary adjustments in your code/ configuration.
- Make sure you identify all the details regarding the configuration (e.g. the test coverage is displayed)
- Once your pipeline succeeds proceed to the next point regarding Continuous Deployment.

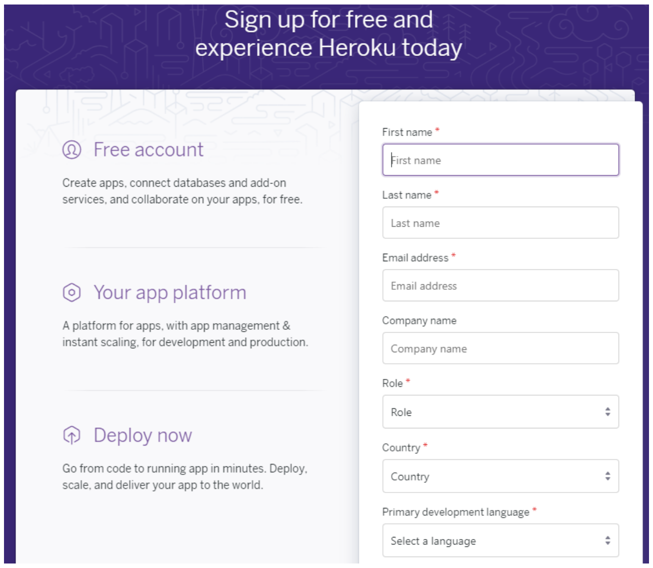
3. Continuous Deployment (CD) – Backend

The continuous deployment aims at delivering as fast as possible the new features added through your code to the deployment servers. This can be configured also as a stage in the .gitlab-ci.yml file, such that if all the previous steps (build, test, and checkstyle) are successful, it may proceed with the deployment of your application on a server. In order to avoid unnecessary deployments, it is recommended to configure the deployment stage such that to be run only for specific branches, when the feature you are working on is completed, and ready to be delivered to the end-user. More details about this will be covered in section 3.2.

3.1. Setting the Application on Heroku

In order to be able to deploy your application, a server instance or a cloud account is required. For this setup, we chose to use the Heroku cloud.

You can create your own account on Heroku for free: https://signup.heroku.com/


Sign up for free and experience Heroku today

Free account

Create apps, connect databases and add-on services, and collaborate on your apps, for free.

Your app platform

A platform for apps, with app management & instant scaling, for development and production.

Deploy now

Go from code to running app in minutes. Deploy, scale, and deliver your app to the world.

First name *	First name
Last name *	Last name
Email address *	Email address
Company name	Company name
Role *	Role
Country *	Country
Primary development language *	Select a language

Figure 10 Heroku Free account

Upon registration you will be able to create a **New Application**. We named our application: "spring-demo-ds2020". For your application, make sure you provide a unique name, that does not already exist.

Then from your Profile, *Account Settings* → *API key* → click reveal and copy the content of your API key.

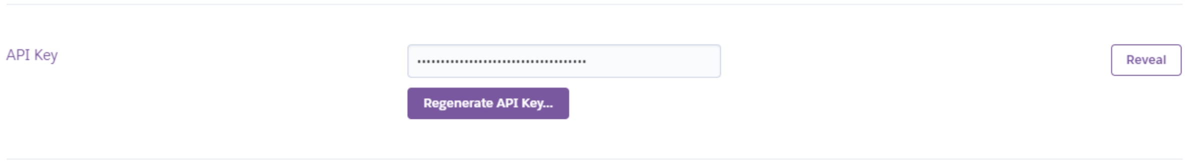
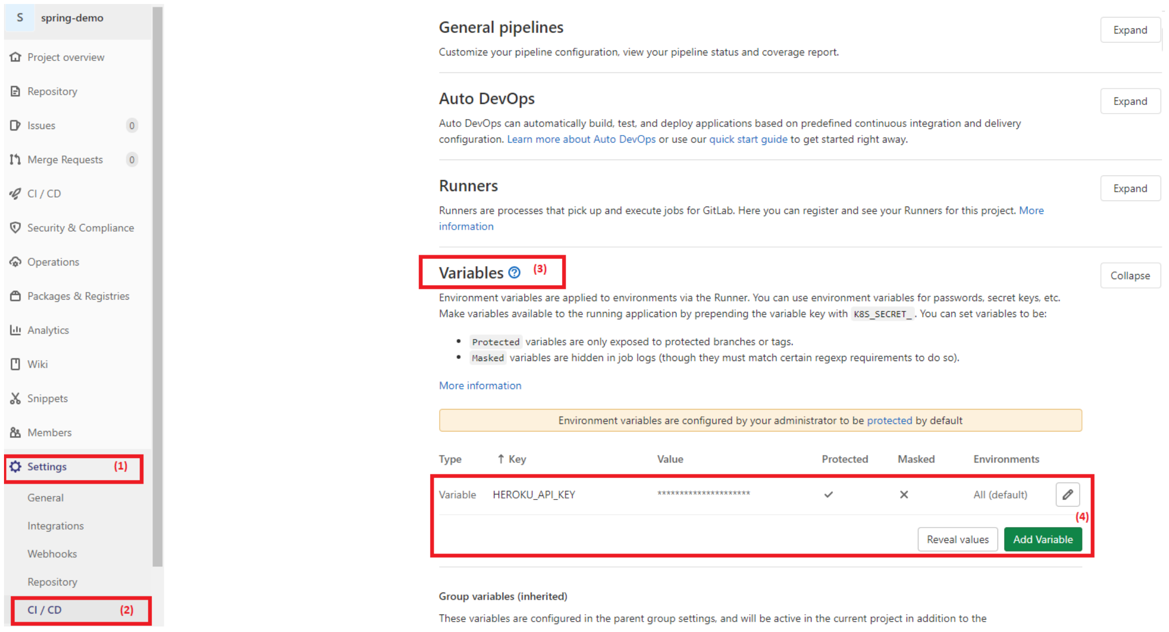


Figure 11 Heroku API key

3.2. Setting the Continuous Deployment phase on Gitlab

The API key needs to be added in the Gitlab's project configuration, in order to be granted access when deploying the application.
For this go to: *Settings* → *CI/CD* → *Variables* → *Add Variable*



General pipelines

Customize your pipeline configuration, view your pipeline status and coverage report.

Auto DevOps

Auto DevOps can automatically build, test, and deploy applications based on predefined continuous integration and delivery configuration. Learn more about Auto DevOps or use our quick start guide to get started right away.

Runners

Runners are processes that pick up and execute jobs for GitLab. Here you can register and see your Runners for this project. More information

Variables

Environment variables are applied to environments via the Runner. You can use environment variables for passwords, secret keys, etc. Make variables available to the running application by prepending the variable key with `K8S_SECRET_`. You can set variables to be:

- `Protected` variables are only exposed to protected branches or tags.
- `Masked` variables are hidden in job logs (though they must match certain regexp requirements to do so).

More information

Environment variables are configured by your administrator to be protected by default

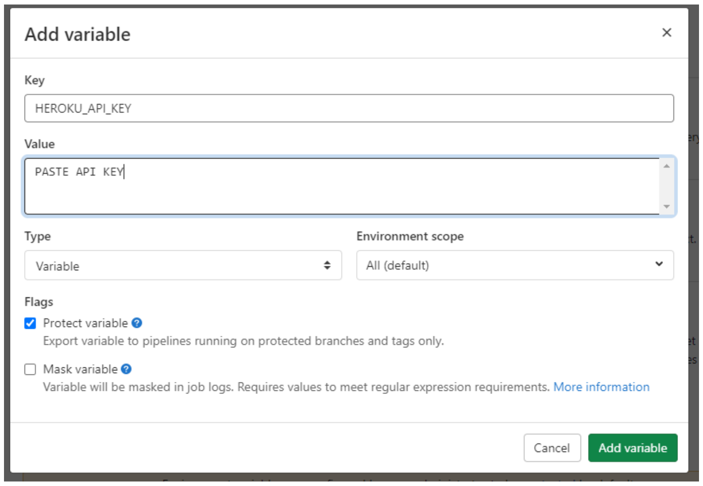
Type	↑ Key	Value	Protected	Masked	Environments
Variable	HEROKU_API_KEY	**********************	✓	✕	All (default)

Group variables (inherited)

These variables are configured in the parent group settings, and will be active in the current project in addition to the

Figure 12 Setup API key in Gitlab

Add the HEROKU_API_KEY variable and paste the API Key that you copied in section 2.1. from Heroku.



Add variable

Key: HEROKU_API_KEY

Value: PASTE API KEY

Type: Variable

Environment scope: All (default)

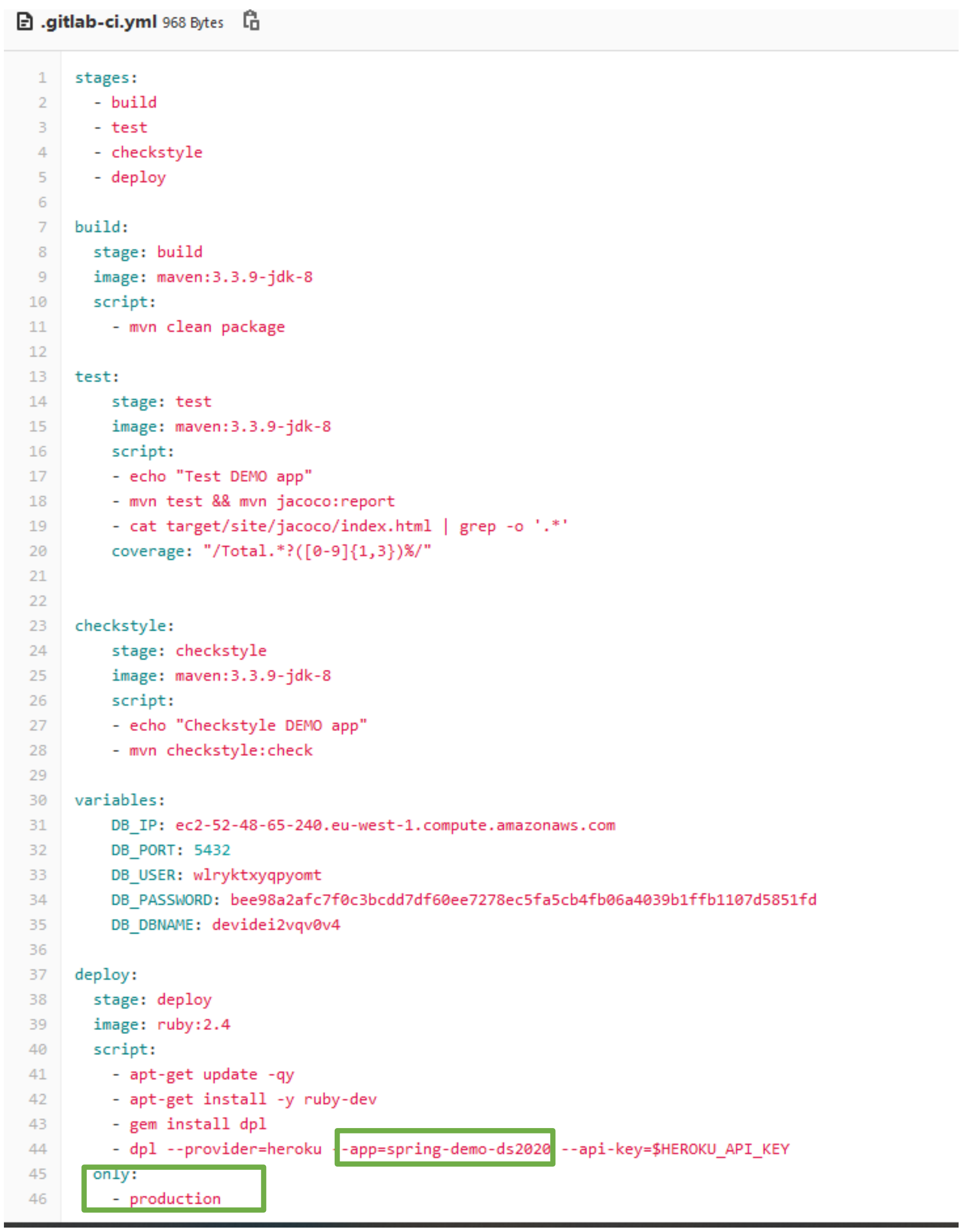
Flags

[✓] Protect variable
Export variable to pipelines running on protected branches and tags only.

[ ] Mask variable
Variable will be masked in job logs. Requires values to meet regular expression requirements. More information

Figure 13 Gitlab Variable for Heroku API ley

Now you can add the deployment stage to your .gitlab-ci.yml file. So the final version of your .gitlab-ci.yml should look like Figure 14, considering for the --app flag, the name you have provided for the Heroku application.



.gitlab-ci.yml 968 Bytes

```yaml
stages:
  - build
  - test
  - checkstyle
  - deploy

build:
  stage: build
  image: maven:3.3.9-jdk-8
  script:
    - mvn clean package

test:
    stage: test
    image: maven:3.3.9-jdk-8
    script:
    - echo "Test DEMO app"
    - mvn test && mvn jacoco:report
    - cat target/site/jacoco/index.html | grep -o '.*'
    coverage: "/Total.*?([0-9]{1,3})%/"


checkstyle:
    stage: checkstyle
    image: maven:3.3.9-jdk-8
    script:
    - echo "Checkstyle DEMO app"
    - mvn checkstyle:check

variables:
    DB_IP: ec2-52-48-65-240.eu-west-1.compute.amazonaws.com
    DB_PORT: 5432
    DB_USER: wlryktxyqpyomt
    DB_PASSWORD: bee98a2afc7f0c3bcdd7df60ee7278ec5fa5cb4fb06a4039b1ffb1107d5851fd
    DB_DBNAME: devidei2vqv0v4

deploy:
  stage: deploy
  image: ruby:2.4
  script:
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --app=spring-demo-ds2020 --api-key=$HEROKU_API_KEY
  only:
    - production
```

Figure 14 FINAL CI/CD configuration

For this setup, we used ruby gems to deploy our spring-boot application on Heroku. As marked in Figure 14, you need to specify the name of your application created in Heroku, and reference the API key variable previously added in Gitlab. Furthermore, you can configure your deployment stage to run only when modifications appear on the specified branches.

3.2.1. Test your solution

Make a modification on your code, locally and then push it to the Gitlab.

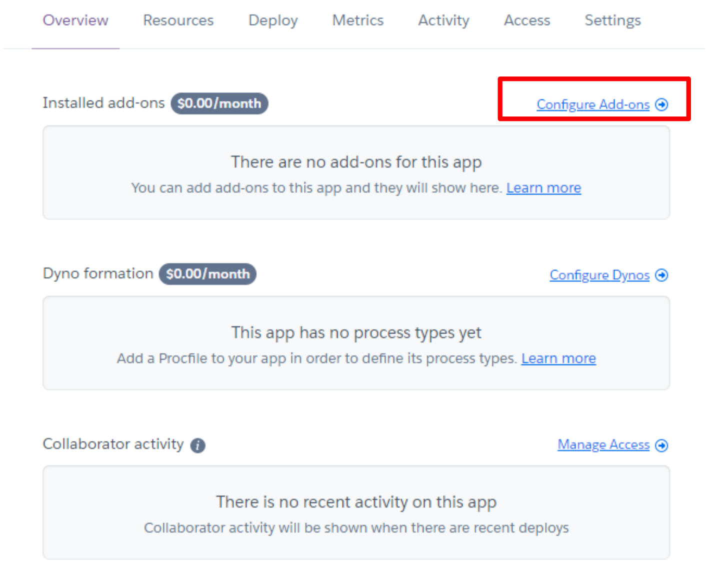
- Check the CI/CD pipelines and make sure that the Gitlab can connect to the Heroku application
- The deployment should be successful, although your spring-boot application will fail to start since it cannot connect yet to a database on the Heroku cloud.
- Make a protected *production* branch and make sure that it runs the deployment only when you merge your developed features in the production branch.

Configuring protected branches

1. Navigate to your project's Settings ➔ Repository.
2. Scroll to find the **Protected branches** section.
3. From the **Branch** dropdown menu, select the **branch** you want to **protect** and click **Protect**. ...
4. Once done, the **protected branch** will appear in the "**Protected branches**" list.

3.3. Setting the Database on Heroku (Postgres)

On your application page in Heroku go to *Overview* and then select *Configure Add-ons*.

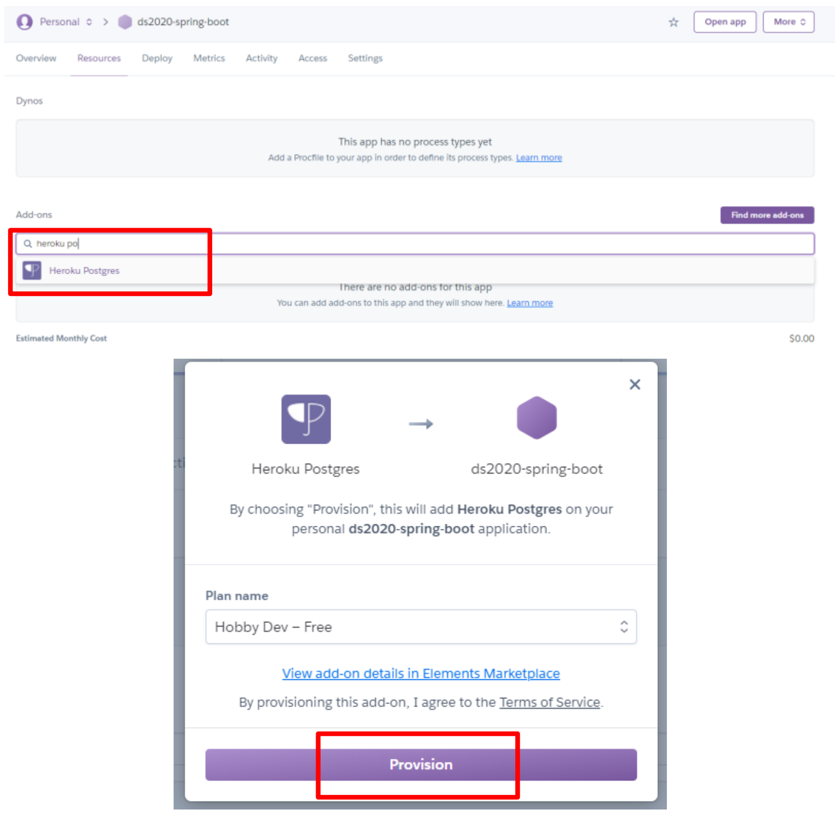


There are no add-ons for this app
You can add add-ons to this app and they will show here. Learn more

This app has no process types yet
Add a Procfile to your app in order to define its process types. Learn more

There is no recent activity on this app
Collaborator activity will be shown when there are recent deploys

Search the Heroku Postgres Add-on (there are other add-ons for databases as well, make sure you chose a free one) and click *Provision.*



This app has no process types yet
Add a Procfile to your app in order to define its process types. Learn more

There are no add-ons for this app
You can add add-ons to this app and they will show here. Learn more

Estimated Monthly Cost $0.00

By choosing "Provision", this will add **Heroku Postgres** on your personal **ds2020-spring-boot** application.

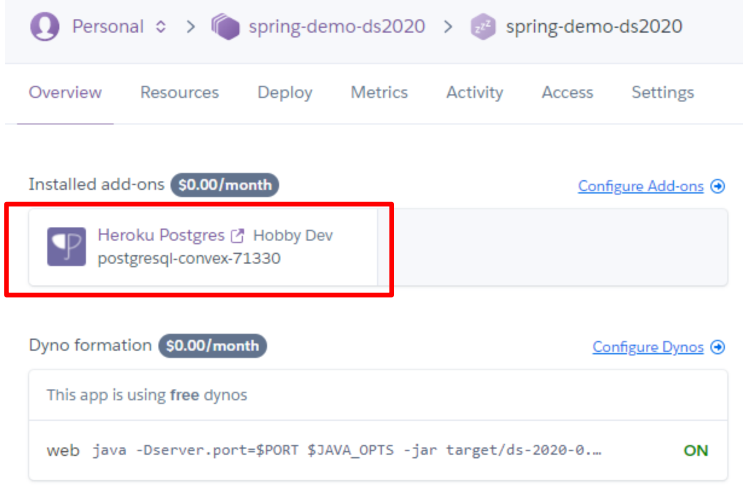
Plan name: Hobby Dev – Free

View add-on details in Elements Marketplace

By provisioning this add-on, I agree to the Terms of Service.

Figure 15 Add database add-on

Under the Overview tag from your application, the Postgres add-on should appear.



Installed add-ons $0.00/month

Heroku Postgres Hobby Dev
postgresql-convex-71330

Dyno formation $0.00/month

This app is using **free** dynos

```
web  java -Dserver.port=$PORT $JAVA_OPTS -jar target/ds-2020-0.…    ON
```

Figure 16 Installed Add ons for application

By selecting your Postgres add-on you will be redirected to the Datastores page, where you can see the details about the Database credentials and other connection information.

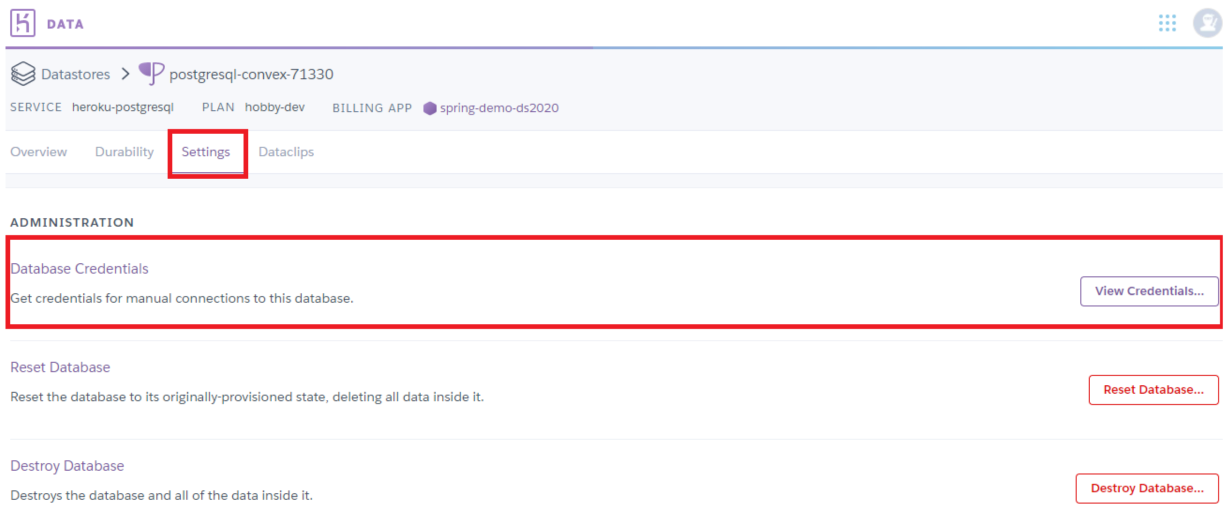
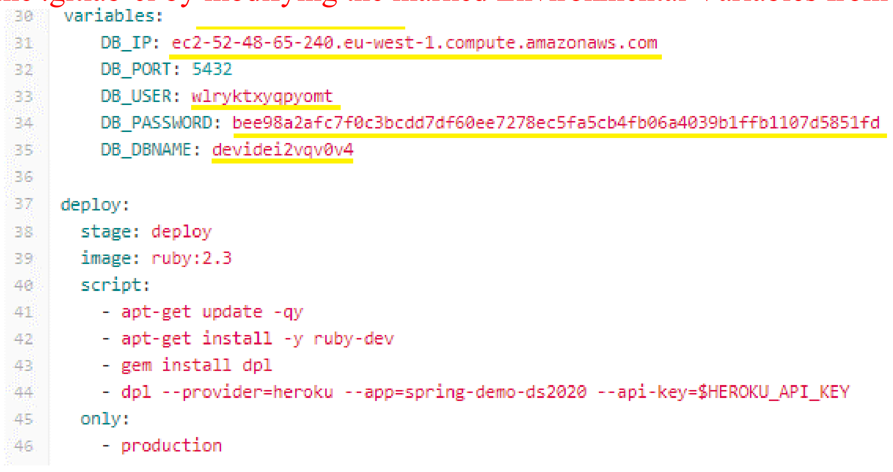

DATA
Datastores > postgresql-convex-71330
SERVICE heroku-postgresql PLAN hobby-dev BILLING APP spring-demo-ds2020
Overview Durability Settings Dataclips
ADMINISTRATION
Database Credentials
Get credentials for manual connections to this database.
View Credentials...
Reset Database
Reset the database to its originally-provisioned state, deleting all data inside it.
Reset Database...
Destroy Database
Destroys the database and all of the data inside it.
Destroy Database...

Figure 17 Database information

OBSERVATION! Based on the Database credentials obtained from the Heroku Administration page, update the .gitlab-ci by modifying the marked Environmental Variables from Figure 18.



```
variables:
    DB_IP: ec2-52-48-65-240.eu-west-1.compute.amazonaws.com
    DB_PORT: 5432
    DB_USER: wlryktxygpyomt
    DB_PASSWORD: bee98a2afc7f0c3bcdd7df60ee7278ec5fa5cb4fb06a4039b1ffb1107d5851fd
    DB_DBNAME: devidei2vqv0v4

deploy:
  stage: deploy
  image: ruby:2.3
  script:
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --app=spring-demo-ds2020 --api-key=$HEROKU_API_KEY
  only:
    - production
```

Figure 18 DB Credentials

3.3.1. Test your solution:

- Using the connection details, you can update your *application.properties* file from your spring boot application to connect to the cloud database you have just created and test it from your IDE locally.
- Trigger the online pipeline on the production branch and check if successful
- From the Heroku page, click on the *Open App* button and you should be redirected to the index of your application deployed on cloud.

IMPORTANT! During the development phase use a local database, since the database created in the Heroku platform has a limited usage plan. Whenever you consider that a feature is completed, switch to your **production** branch.

4. CI/CD – Frontend

For the Frontend application, the same principles apply when setting the CI/CD pipeline. At [2] you can find a React application configured to be built and deployed on Heroku.

4.1. Test your solution:

- Configure your own frontend application and supply the necessary connection details (hostname in hosts.js) so that the frontend application can connect to your backend application.

References

[1] https://gitlab.com/ds_20201/spring-demo
[2] https://gitlab.com/ds_20201/react-demo
[3] https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml