MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DISTRIBUTED SYSTEMS

CI/CD Deployment using Docker on 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 using Dockers. You can use the source code provided in [1] and [3] and setup your own repository on Gitlab and following the instructions and the exercises from *"Test your solution"*. The docker-configuration is found on the ***docker-production*** branch in the specified repositories. By the end of the laboratory you should have your own backend and frontend application configured to run both the CI and the CD pipeline using Dockers.

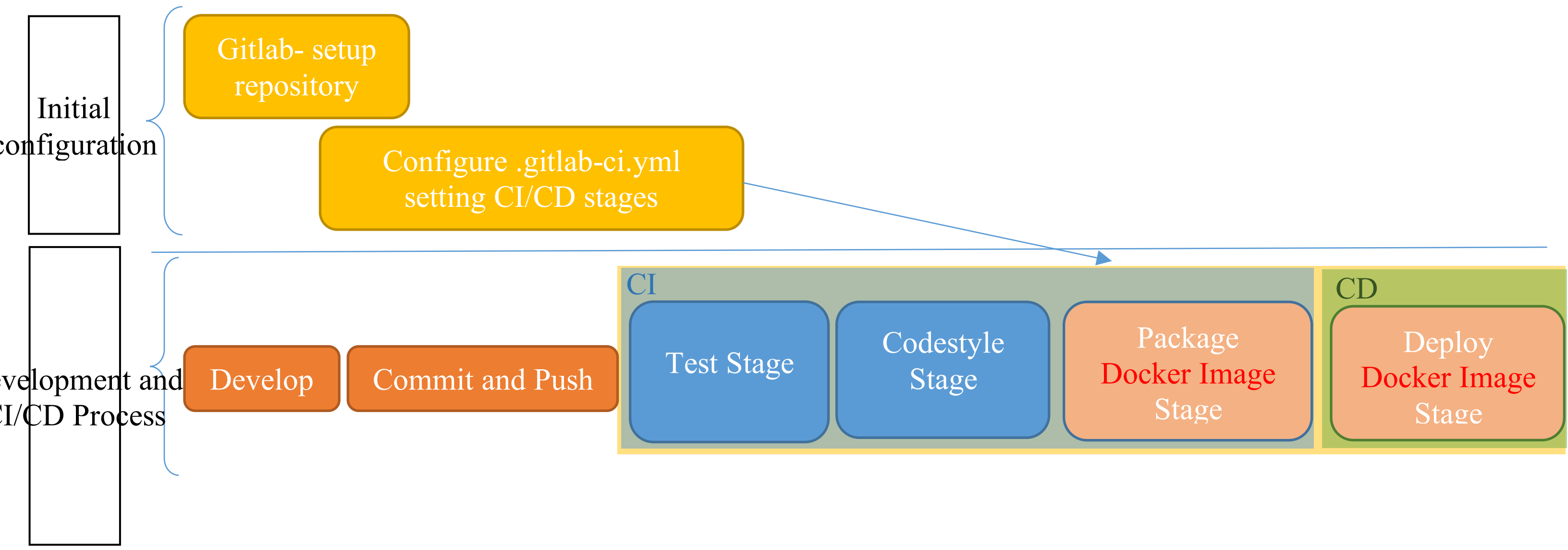


Figure 1. CI/CD Pipeline

With respect to the previous CI/CD setup established for the first assignment on the **production** branch of the repositories, the docker-based CI/CD replaces the Build Phase and Deploy Phase with docker instructions that will be exemplified in the following two chapters.
The Heroku cloud allows for the **free account a 60 second boot time and 512MB memory**. For this reason, some extra measures are considered when configuring the Docker images in order to improve the application startup resources consumption.

2. Docker

2.1. What is Docker? Why choosing Docker over VMs?

The Operating System divides the computer memory in several sections, where the **Kernel space** and the **User space** are most important, as shown in Figure 2. The **Kernel Space** is the portion of memory where privileged operating system kernel processes are executed, while the User Space contains unprivileged processes. The separation is performed using a set of privileges. Programs or processes are run in user mode and are sandboxed, meaning that they are isolated from other processes from the memory point of view and cannot have complete access to the computer's memory, disk storage, network hardware, and other resources.

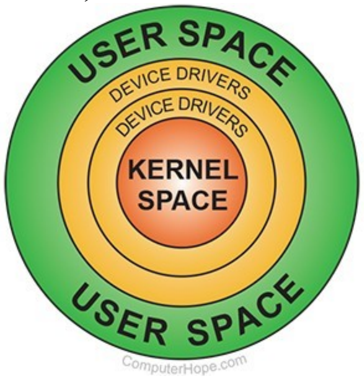


Figure 2. Computer Memory privileges separation (Source [8]).

Starting with 1970, IBM and later other companies started developing special software called **Hypervisor**, used to create and run Virtual Machines (VMs) [9]. A VM is an emulation of a computer system, based on a computer architecture and providing the functionality of a physical computer [15]. Several VMs with different hardware requirements and guest OSes can be run on a host computer. Multiple instances of a variety of operating systems may share the virtualized hardware resources: for example, Linux, Windows, and macOS instances can all run on a single physical x86 machine. The hypervisor runs in **Kernel mode**, while the **guest OS** runs in **user mode**, thus theoretically being **sandboxed** from the host OS.

As opposed to virtualization, **Docker** is a container-based technology where containers are running as processes in the user space of the operating system. Docker originally used LinuX Containers (LXC), but later switched to runC (formerly known as libcontainer), which runs in the same operating system as its host. This allows it to share a lot of the host operating system resources. At the low level, a container is just a set of processes that are isolated from the rest of the system, running from a distinct image that provides all files necessary to support the processes. It is built for running applications. In Docker, the containers running share the host OS kernel.

Table 1. VM and Docker comparison [10,11,12]

Features	VM	Docker
Host OS	Any	Linux – based (if installed on Windows it installs a VM with Linux)
Guest OS	Any	Linux – based (it uses the kernel of the host operating system)
Sandboxing	Full isolation	Can access host through shared filesystem (such as docker-volume)
HW Resource requirements	High (similar to the physical machine emulated)	Low (many containers can run on the same physical machine)

2.2. How to install Docker

Recommended to use Linux. Docker for Windows seems to create a VM with Linux on it, on which it runs Docker, inside which it then runs containers. So much indirection might lead to problems in the future (such as managing volumes). Furthermore, **docker** commands need **privileged** access, using **sudo** command. Follow the tutorial here and install Docker CE [7]: https://docs.docker.com/install/linux/docker-ce/ubuntu/

2.3. Docker container lifecycle

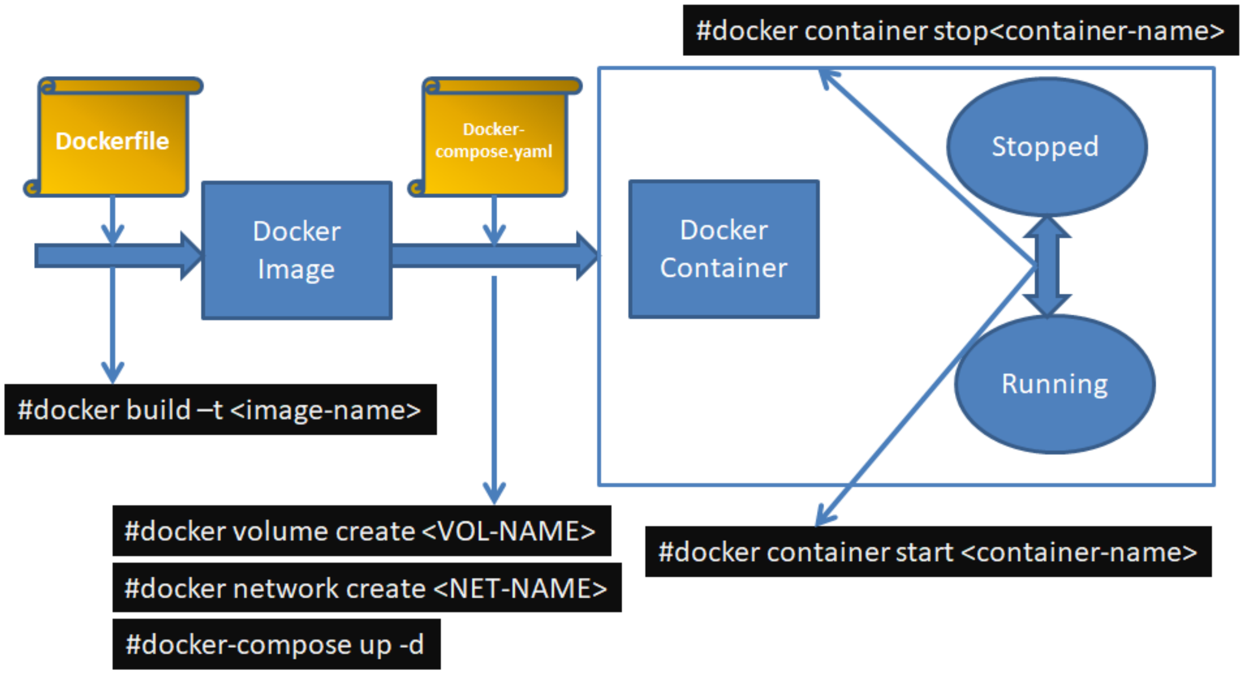

#docker container stop<container-name>
Dockerfile
Docker-compose.yaml
Docker Image
Docker Container
Stopped
Running
#docker build –t <image-name>
#docker volume create <VOL-NAME>
#docker network create <NET-NAME>
#docker-compose up -d
#docker container start <container-name>

Figure 3. Docker lifecycle and basic commands

The first step would be to create an empty folder for each docker container you want to create. This folder should contain at least 2 files, with exactly these names (since only these are recognized by docker). The files will be detailed in the following sections.

- Dockerfile – the description of the image
- Docker-compose.yaml – the description of the container.

Docker REGISTRY

The Docker registry [13] is a server application that stores and distributes Docker images. Almost all custom images that will be build are based on an existing image that already exists in the Docker Registry [14]. Images can also build from scratch, without inheriting any parent image.

Create Docker Image

The Docker image is described by a *Dockerfile*. An image can be created either by inheriting a parent image, or by creating a base image from scratch [14]:

- **Inherit a parent image**: the new image will customize an existing image (parent image) from Docker Registry, by referencing it using the FROM directive at the beginning of the Dockerfile. All the other instructions in the docker file modify the parent image.
- **Create a base image**: a base image is created by using the FROM scratch directive at the beginning of the Dockerfile.

Thus, a docker image can either be used directly from an online repository (e.g. the Docker REGISTRY) or it can be customized starting from a parent image by describing it in a Dockerfile and issuing the following command in the terminal:

```
#docker build –t <image-name>
```

An example for a Dockerfile for Nodejs server (downloaded from Docker REGISTRY) customized for deploying an application is the following:

```
FROM node:8
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
CMD npm start
EXPOSE 3000
```

By issuing the docker build command, an image with the name <image-name> will be created. At this stage, the image is stored locally and can be viewed using the command:

```
#docker image ls
```

Create Docker Volume

To create a docker volume you just have to run:

```
#docker volume create <vol_name>.
```

The information from the volume will be stored at */var/lib/docker/volumes/<vol_name>/_data*. You can copy any files of interest directly here and they will appear in the container in the specified folder (see Figure 4).

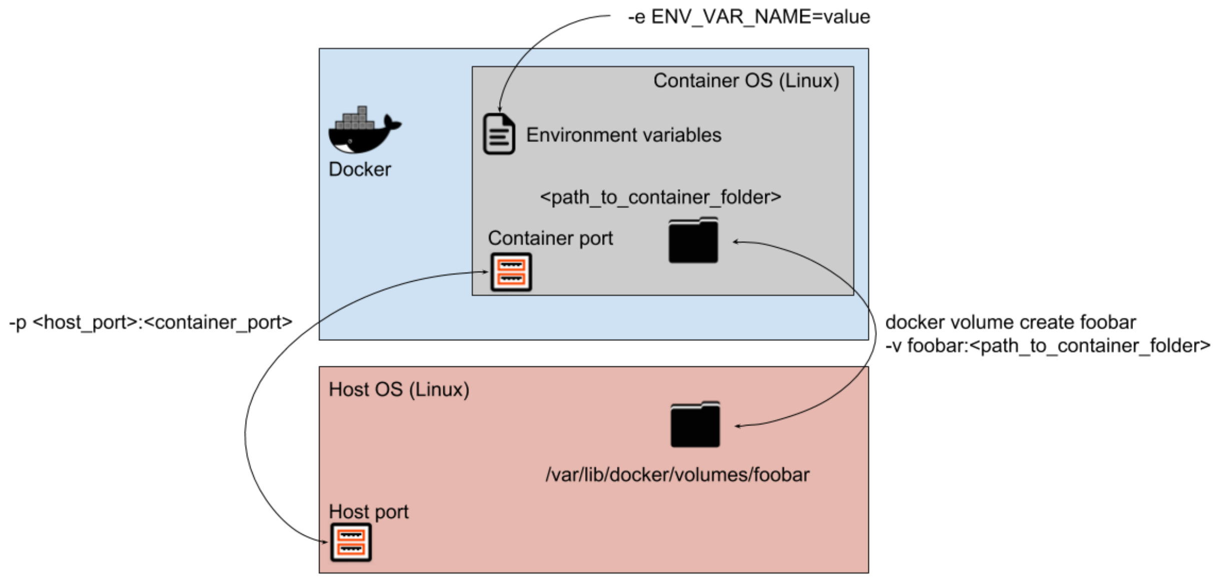

-e ENV_VAR_NAME=value
Docker
Container OS (Linux)
Environment variables
<path_to_container_folder>
Container port
-p <host_port>:<container_port>
docker volume create foobar
-v foobar:<path_to_container_folder>
Host OS (Linux)
/var/lib/docker/volumes/foobar
Host port

Figure 4. A docker container and communication with the host (port mapping and shared folders using volumes)

To see all available volumes, you can run docker volume ls. To remove a volume you can run *docker volume rm <vol_name>*.

Configure Network

A virtual network can be created between the containers. Each network has a name and a IP address. To create a network, the following command is used:

```
#docker network create <NET-NAME>
```

Create Container

There are 3 basic things that we want to synchronize between a docker container and the host:

- ports: map a container's port to a host's port (to be able to access the application from outside)
- volumes: map a container's folder to a host's Docker volume (to be able to persist information and do backups)
- environment vars: set the container's environment variables that can be used by the application for various configurations

You can use docker-compose files to write the configurations you want for a docker container. A docker-compose file MUST be named docker-compose.yaml, and for the syntax you can check the existing ones or the internet.

The docker-compose files for the most used images are already created, and you can find them in Docker-compose Scripts folder at this location (mysql, cassandra, tomcat, gitlab).

In order to run/create a docker container you have to move this docker-compose.yaml file to the desired computer, cd to its location and run:

```
#docker-compose up –d
```

Make sure the appropriate docker volumes are created before running the above command (e.g. tomcat volume for tomcat image, check the used volumes in each docker-compose file in particular).

Containers created and started with this are started and stopped with the classic Docker commands (start, stop, restart).

Check Running container and logs

The containers that are running can be checked with

```
#docker ps
```

The logs of a container can be accessed using

```
#docker logs –f <container-name>
```

Stop/Remove Container and Image

Containers and corresponding images can be stopped and removed using the following commands (in this order):

```
#docker container stop <container-name>
#docker container rm <container-name>
#docker image rm <image-name>
```

Enter terminal of a container

One can enter the terminal of a container using the following command:

```
#docker exec –it <container-name> /bin/bash
```

Furthermore, using the instruction *docker exec –it <container-name>*, other commands to the container can be appended.

2.4. Docker Commands Summary

```
Interogate running containers:
> docker ps

Interogate existing volumes:
> docker volume ls

Stop and Remove running containers:
> docker stop {name}
> docker rm {name}
```

```
Remove existing volume:
> docker volume rm {volume-name}

Use docker-compose.yaml files, if the container needs a volume create it first with:

> docker volume create {volume-name}

Then start the docker container using:
> docker-compose up -d


Volumes location on host:
/var/lib/docker/volumes/

Access container bash:

docker exec -it [container-id] /bin/bash
```

2.5. DOCKER deploy example

In this section we will exemplify the deployment on Docker of the Spring demo application and React application from tutorials for Assignment 1. The code can be downloaded from:

- Spring Application [1]: git clone https://gitlab.com/ds_20201/spring-demo.git
- React Application [3]: git clone https://gitlab.com/ds_20201/react-demo

We suppose the database connection to the PostgreSQL deployed on Heroku in the previous laboratory session.

2.5.1. Deployment of Spring application

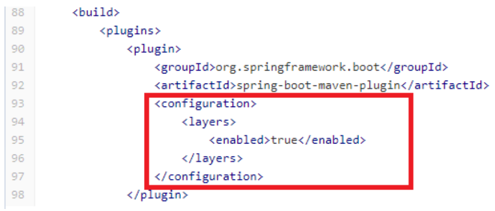
In order to be able to build a custom docker image containing your application's executable code a Dockerfile must be configured in the root directory of the project.
A Dockerfile and docker-compose.yml files are already available in the Spring demo if you switch the branch to ***docker_production***. Otherwise, create a new branch and create yourselves the files specified in the laboratory work.

```
#git fetch -a
#git branch -a
#git checkout docker_production
```

The Dockerfile used for building the image for our application is presented in Figure 6.
Starting with line 1, an intermediate maven image called builder is configured. This is used in order to build the executable of the application from the source code. Thus, firstly the source code is copied in the temporary image (src, pom.xml and checkstyle.xml). Normally, a Spring Boot application can be started using this .jar file. **However, due to the boot resources limitation, we**

have considered a more optimal approach, by using layered Jars. More details about the Layered Jars and the reasons for using them can be found at [2].

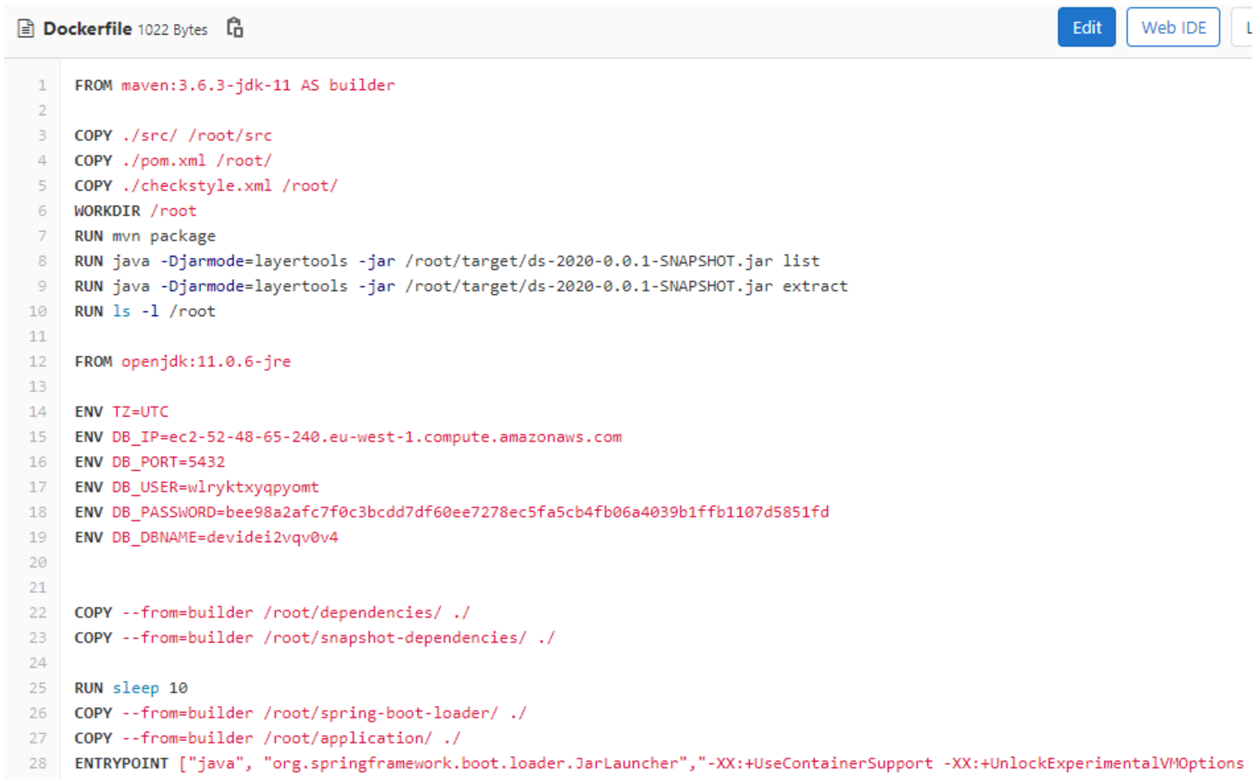
1. In order to specify that a layered jar is required, firstly you need to modify lines 93-97 from pom.xml file, and add the following configuration, specifying that layers are enabled.



```
88        <build>
89            <plugins>
90                <plugin>
91                    <groupId>org.springframework.boot</groupId>
92                    <artifactId>spring-boot-maven-plugin</artifactId>
93                    <configuration>
94                        <layers>
95                            <enabled>true</enabled>
96                        </layers>
97                    </configuration>
98                </plugin>
```

Figure 5. Section from pom.xml of spring-demo

2. The ***mvn package*** is run (Figure 6 line 7) in order to obtain the application's layered .jar file from the source code.
3. The layers of the created .jar file are listed using command at line 8
4. The layers of the created .jar file are extracted using command at line 9
5. Lines 14-19 from the Dockerfile presented in Figure 6 contain the DB credentials from the PostgreSQL deployed in Heroku in the previous laboratory session.



```
1   FROM maven:3.6.3-jdk-11 AS builder
2
3   COPY ./src/ /root/src
4   COPY ./pom.xml /root/
5   COPY ./checkstyle.xml /root/
6   WORKDIR /root
7   RUN mvn package
8   RUN java -Djarmode=layertools -jar /root/target/ds-2020-0.0.1-SNAPSHOT.jar list
9   RUN java -Djarmode=layertools -jar /root/target/ds-2020-0.0.1-SNAPSHOT.jar extract
10  RUN ls -l /root
11
12  FROM openjdk:11.0.6-jre
13
14  ENV TZ=UTC
15  ENV DB_IP=ec2-52-48-65-240.eu-west-1.compute.amazonaws.com
16  ENV DB_PORT=5432
17  ENV DB_USER=wlryktxyqpyomt
18  ENV DB_PASSWORD=bee98a2afc7f0c3bcdd7df60ee7278ec5fa5cb4fb06a4039b1ffb1107d5851fd
19  ENV DB_DBNAME=devidei2vqv0v4
20
21
22  COPY --from=builder /root/dependencies/ ./
23  COPY --from=builder /root/snapshot-dependencies/ ./
24
25  RUN sleep 10
26  COPY --from=builder /root/spring-boot-loader/ ./
27  COPY --from=builder /root/application/ ./
28  ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher","-XX:+UseContainerSupport -XX:+UnlockExperimentalVMOptions
```

Figure 6 Dockerfile for Spring Boot Application

The steps to obtain the DB credentials are shown in the following figure and can be also found in "*CI/CD Tutorial and Deployment on cloud (Heroku Cloud)*". First, login in the Heroku cloud, access your spring-demo project, access the Resources section and the Configure Add-Ons. Select the Heroku Postgres Add-on, go to Settings and Database Credentials, as shown in Figure 7.

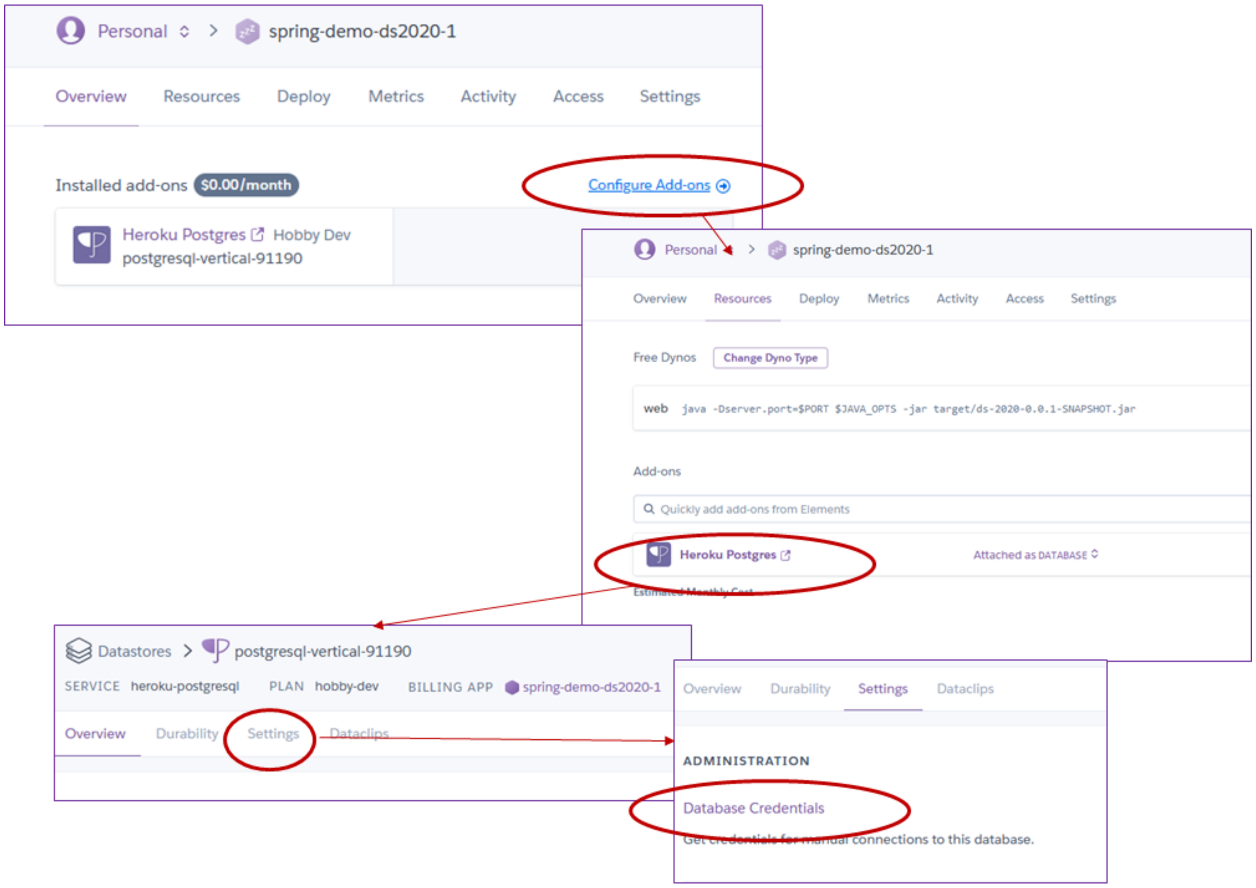
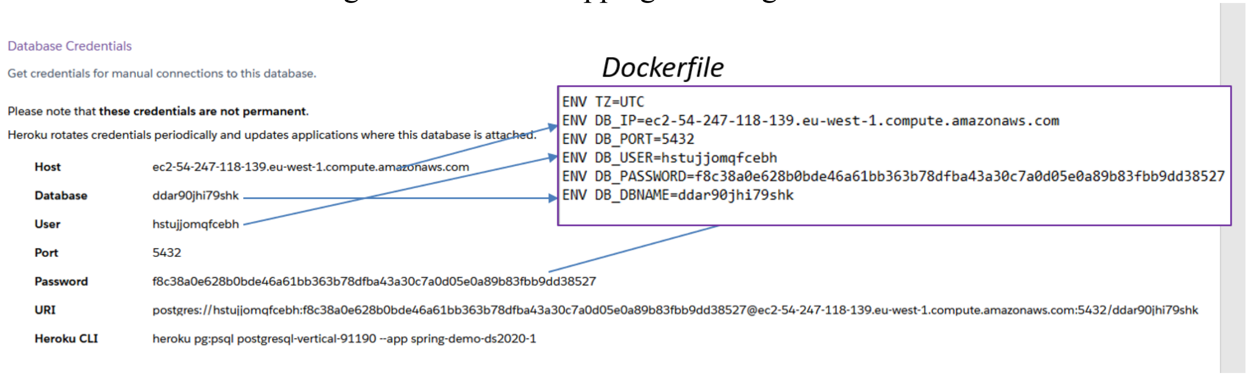


Figure 7. Steps to obtain DB credentials.

The credentials for your database connection will be displayed. Replace them in lines 16-19 from the Dockerfile shown in Figure 6 with the mapping from Figure 8.



Database Credentials

Get credentials for manual connections to this database.

Please note that **these credentials are not permanent**.

Heroku rotates credentials periodically and updates applications where this database is attached.

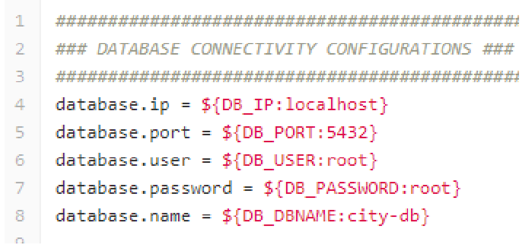
Host	ec2-54-247-118-139.eu-west-1.compute.amazonaws.com
Database	ddar90jhi79shk
User	hstujjomqfcebh
Port	5432
Password	f8c38a0e628b0bde46a61bb363b78dfba43a30c7a0d05e0a89b83fbb9dd38527
URI	postgres://hstujjomqfcebh:f8c38a0e628b0bde46a61bb363b78dfba43a30c7a0d05e0a89b83fbb9dd38527@ec2-54-247-118-139.eu-west-1.compute.amazonaws.com:5432/ddar90jhi79shk
Heroku CLI	heroku pg:psql postgresql-vertical-91190 --app spring-demo-ds2020-1

Dockerfile

```
ENV TZ=UTC
ENV DB_IP=ec2-54-247-118-139.eu-west-1.compute.amazonaws.com
ENV DB_PORT=5432
ENV DB_USER=hstujjomqfcebh
ENV DB_PASSWORD=f8c38a0e628b0bde46a61bb363b78dfba43a30c7a0d05e0a89b83fbb9dd38527
ENV DB_DBNAME=ddar90jhi79shk
```

Figure 8. Mapping for Environment variables from Dockerfile to Heroku interface

Starting with line 12, a JDK 11 image is used and the layers obtained in the previous temporary image are copied in this image (lines 22, 23, 26, 27). Furthermore, the details for the DB connections can be specified in the Dockerfile as environmental variables. In this way, the source code will not contain the connection details but will be able to read them from the Environmental Variables set in the Dockerfile. This is possible by having the right configuration in your spring-boot ***application.properties*** file. Each variable form the ***application.properties*** has a format of `${ENV_VAR_NAME: default_value}` specifying that, if the ENV_VAR_NAME is found in the local environmental variables, then that value is considered, otherwise the default value is considered. On one hand, in the development mode (when the project is setup in your IDE) there are no environmental variables set, so the default values are considered. On the other hand, when the docker image is launched, the environmental variables are set (Figure 9 lines 15-19) so the specified values are considered, and the default ones are ignored.



```
###############################################
### DATABASE CONNECTIVITY CONFIGURATIONS ###
###############################################
database.ip = ${DB_IP:localhost}
database.port = ${DB_PORT:5432}
database.user = ${DB_USER:root}
database.password = ${DB_PASSWORD:root}
database.name = ${DB_DBNAME:city-db}
```

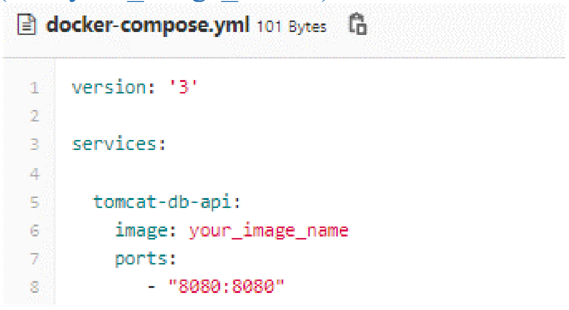
Figure 9. Application Properties section from spring-demo

Line 28 from the Dockerfile from Figure 6, specifies the command with which the newly created image should be launched. As noticed, there are several options set in order to optimize the resources consumption during boot time.
Remember to set the spring.jpa.hibernate.ddl-auto property to create/validate/update according to your database structure and contents.

2.5.1.1. Test your solution

Before continuing your configuration on the Gitlab repository, make sure that your ***Dockerfile*** written by you is correct and that the obtained image can run successfully.
For this follow the instructions:

1. Create a ***docker-compose.yml*** file in the root of your project, containing the following lines. The name of the image must correspond to the name given as argument to the build command from step 2 (i.e. "your_image_name")



docker-compose.yml 101 Bytes

```
version: '3'

services:

  tomcat-db-api:
    image: your_image_name
    ports:
       - "8080:8080"
```

Figure 10. Docker-compose file of spring-demo image

2. Build your image using:
   - *docker build -t your_image_name .*
3. Start your image:
   - *docker-compose up -d*

4. Access your deployed application at *http://localhost:8080*. Furthermore, other endpoints of the application should be accessible, such as *http://localhost:8080/person*, returning the list of persons stored in the DB.

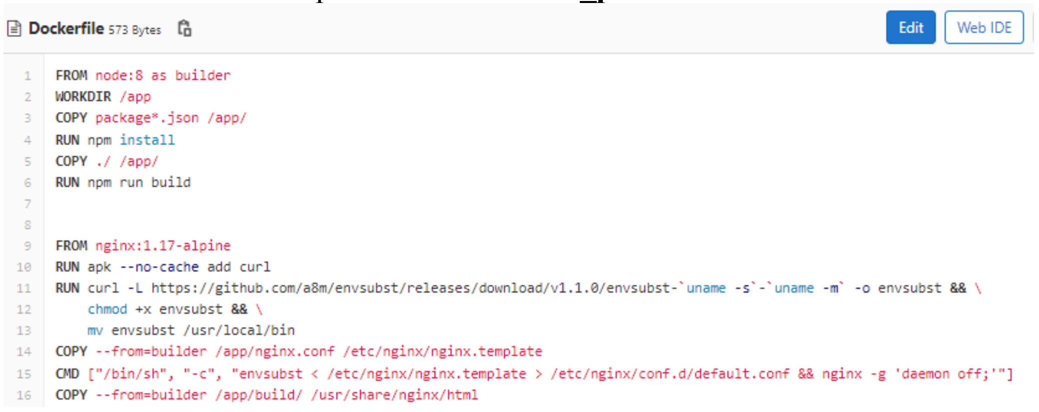
If everything is successful, you can push your newly created files on your repository (create a new branch like the example given ***docker_production*** branch) and proceed with the Gitlab configuration.

2.5.2. Deployment of React application

For the Frontend application, the same principles apply when setting the CI/CD pipeline. At [3] you can find a React application configured to be built and deployed on Heroku using dockers. The docker based CI/CD is configured on the **docker_production** branch.

Observations:

- Check the Dockerfile exposed on the **docker_production** branch:

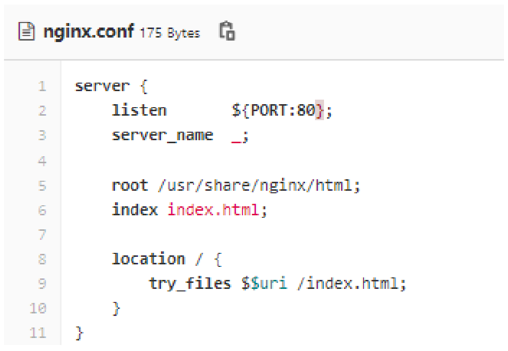


```
FROM node:8 as builder
WORKDIR /app
COPY package*.json /app/
RUN npm install
COPY ./ /app/
RUN npm run build


FROM nginx:1.17-alpine
RUN apk --no-cache add curl
RUN curl -L https://github.com/a8m/envsubst/releases/download/v1.1.0/envsubst-`uname -s`-`uname -m` -o envsubst && \
    chmod +x envsubst && \
    mv envsubst /usr/local/bin
COPY --from=builder /app/nginx.conf /etc/nginx/nginx.template
CMD ["/bin/sh", "-c", "envsubst < /etc/nginx/nginx.template > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"]
COPY --from=builder /app/build/ /usr/share/nginx/html
```

Figure 11. Dockerfile for docker production branch

Same approach is used as for the Maven project. In the first stage the application is built in an intermediate node image, while the built results are copied in the final nginx image. The Envsubst plugin is installed in order to make possible the parametrization of the nginx scripts with Environmental Variables. More details about this at [6].

- A nginx.conf file must be added in the root directory in order to specify the configuration for the nginx server. Here the port of the server is passed as an environmental variable by the Heroku cloud (line 2).



nginx.conf 175 Bytes

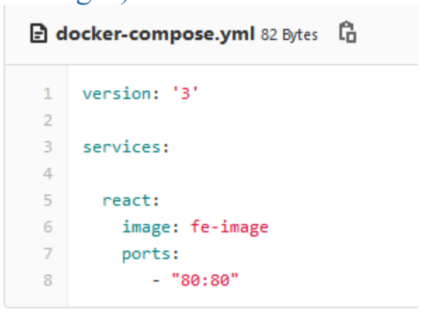
```
server {
    listen       ${PORT:80};
    server_name  _;

    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $$uri /index.html;
    }
}
```

Figure 12. nginx.conf configuration file

2.5.2.1. Test your solution

Before continuing your configuration on the Gitlab repository, make sure that the Dockerfile written by you is correct and that the obtained image can run successfully.
For this follow the instructions:

1. Create a ***docker-compose.yml*** file in the root of your project, containing the following lines. The name of the image must correspond to the name given as argument to the build command from step 2 (i.e. “fe-image”)



docker-compose.yml 82 Bytes

```
version: '3'

services:

  react:
    image: fe-image
    ports:
      - "80:80"
```

Figure 13. Docker-compose configuration file for React app

2. Build your image using:
   - *docker build -t fe-image .*
3. Start your image:
   - *docker-compose up -d*
4. Access your deployed application at *http://localhost*

If everything is successful, you can push your newly created files on your repository (create a new branch like the example given ***docker_production*** branch) and proceed with the Gitlab configuration.

3. Project Deployment

The goal of this tutorial is to deploy the software stack in Docker containers, to handle the heterogeneity of the platforms and to ease the migration to the Heroku platform.

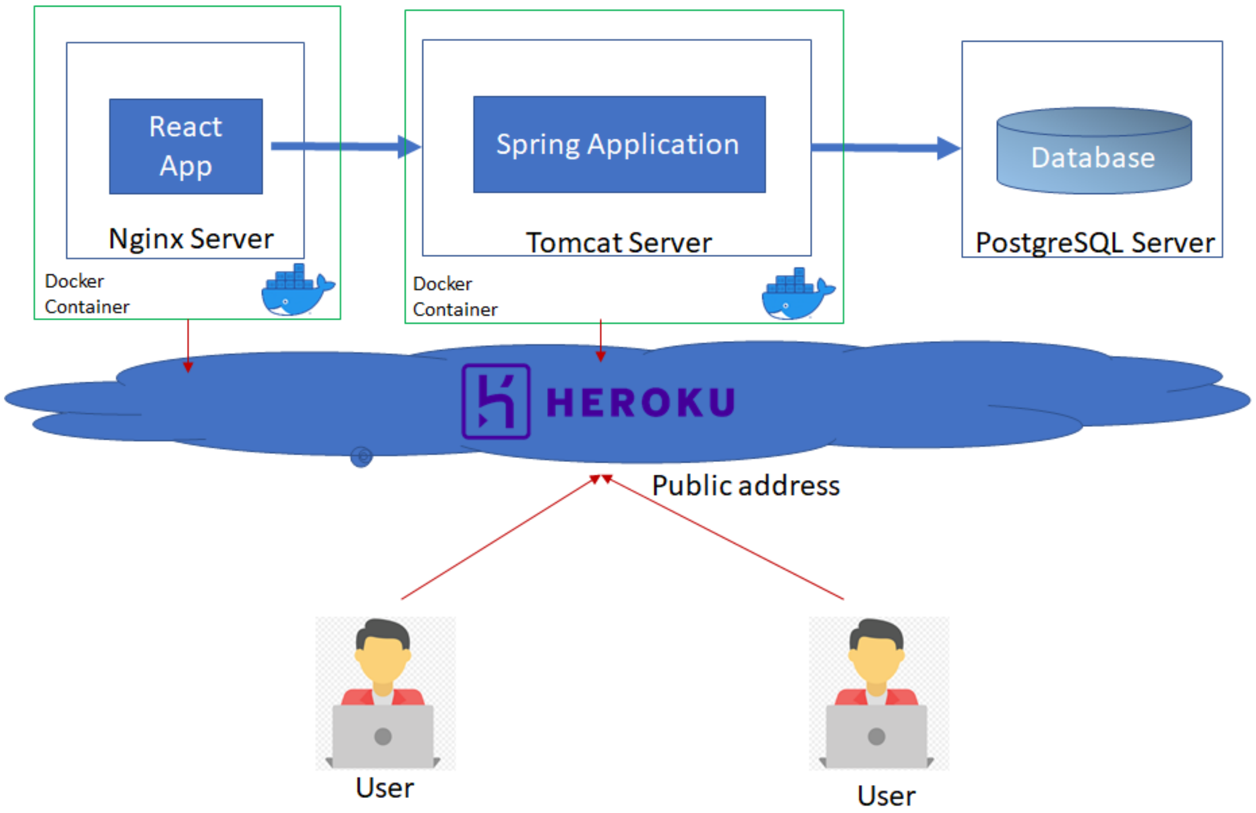


Figure 14. Project Deployment Diagram

Each module of the application, namely the database server and the backend server will be deployed in Docker containers that will be hosted on the Heroku cloud, thus eliminating the need of custom scripts used for application deployment in the previous laboratory tutorial “CI/CD Tutorial and Deployment on cloud (Heroku Cloud)”. The database is deployed in an instance of PostgreSQL server as shown in the previous laboratory work.

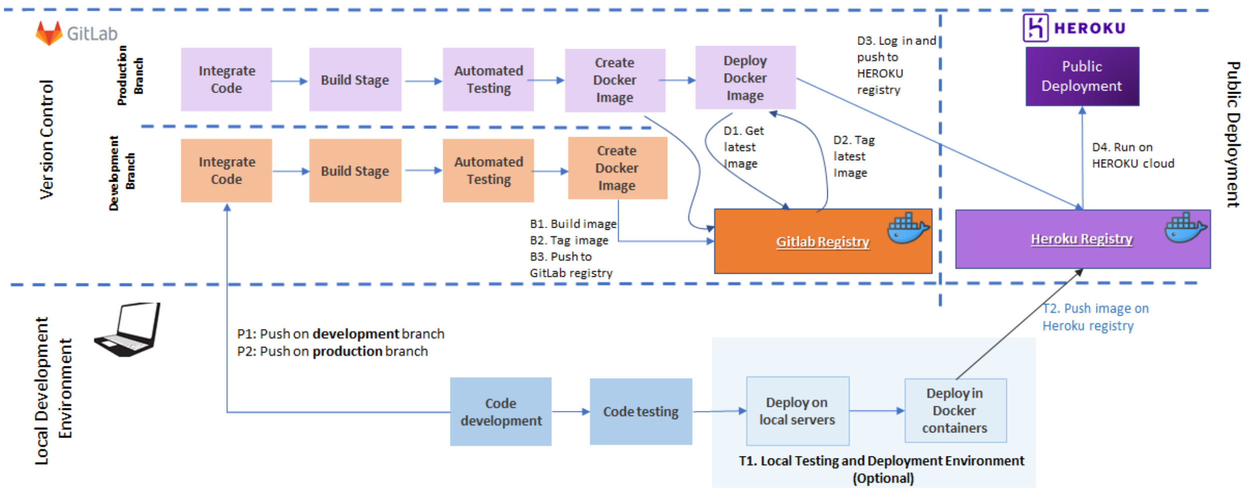


Figure 15. Project Deployment Pipeline

The steps that are involved in deploying the application on the Heroku cloud are depicted in Figure 15 and described in Table 2:

Table 2. Project Deployment Pipeline Steps

Step	Description	Detailed in section
P1	Develop code locally. Push on development branch while adding new features.	Previous laboratory work: "CI/CD Tutorial and Deployment on cloud"
P2	When a stable version is reached, merge the development branch with the production branch and push the code.	
T1 (OPTIONAL)	Test that the solution runs without problems on a local test environment. Initially deploy on web servers, then deploy on local Docker containers.	Section 2.5.
T2 (OPTIONAL)	Test the connection with the Heroku registry. Push the image to Heroku registry and then perform a manual deploy on Heroku.	Section 4.2.1.
B1	Build the image from the code pushed and built on one of the branches from GitLab.	Section 4.1.1.
B2	Tag the built image	
B3	Push the image on GitLab registry	
D1	Login to GitLab registry and get the latest image from the production branch (other images may exist from building the development branch).	Section 4.2.2.
D2	Tag the latest image	
D3	Login and push the image to the Heroku registry	
D4	Trigger the automatic deploy of the image on Heroku cloud	

4. GitLab CI/CD using Docker

This section covers the steps needed to deploy the Spring-demo app in Docker containers on the Heroku cloud:

- ➢ The first part of this section covers the setup to automate the process of creating images by the CI pipeline from GitLab and save them in GitLab Registry.
- ➢ The second part of this section covers the setup to push the images to Heroku cloud by moving them to Heroku registry and them triggering a deploy.

4.1. GitLab CI Automatic Build Docker Image

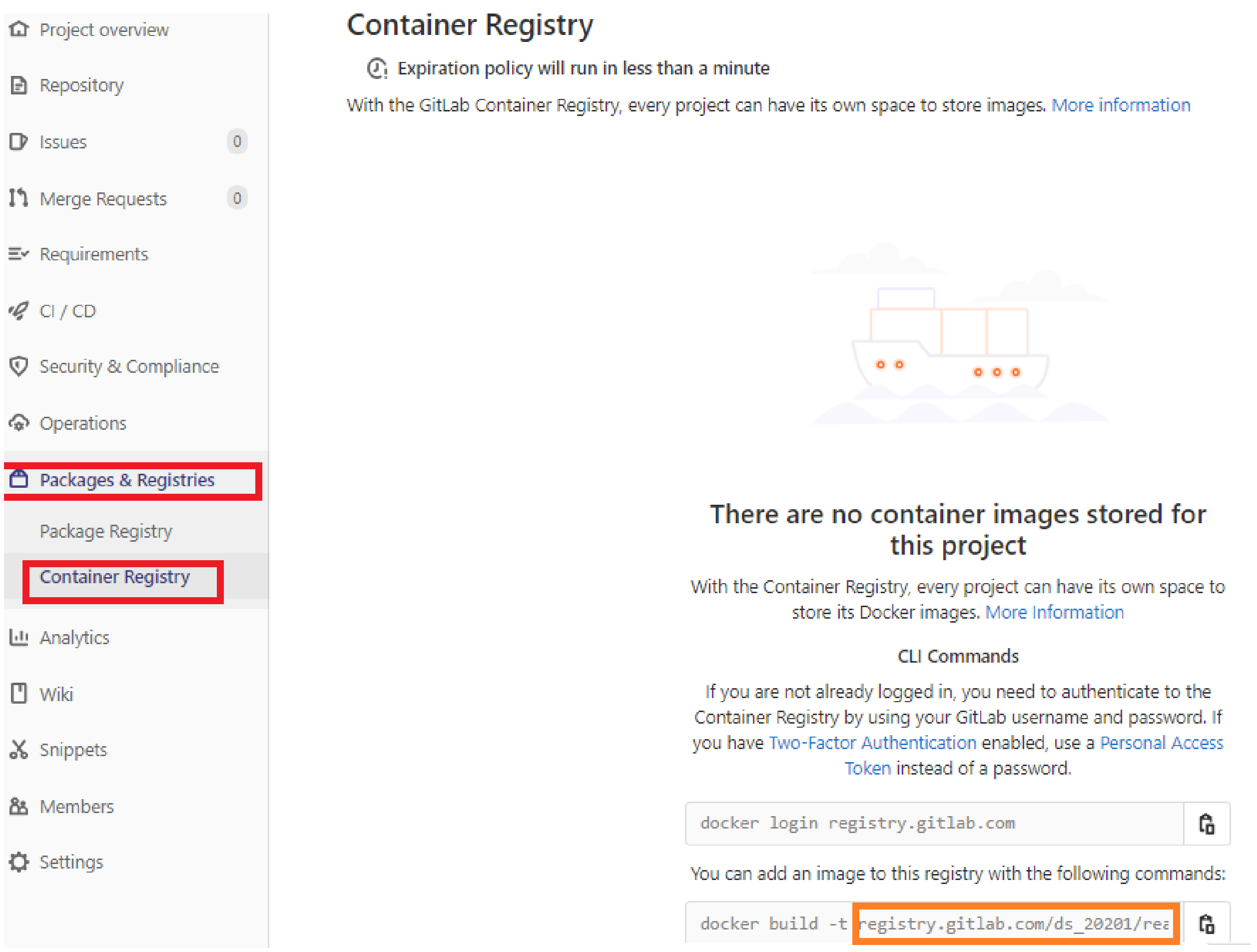
Consider the following setup on your ***docker_production*** branch. Set your branch as protected by going on GitLab to *Settings* → *Repository* → *Protected Branches.*

4.1.1. Configure Gitlab

In your repository, go to *Settings* → *CI/CD* → *Variables* and add the following 2 variables:

• **Key**: CI_REGISTRY	**Value**: registry.gitlab.com
• **Key**: CI_REGISTRY_IMAGE	**Value**: **registry.gitlab.com/group_id/repo_id**

!!! For the Value required for **CI_REGISTRY_IMAGE,** please go to *Packages and Registries* → *Container Registry* and copy the image name provided in the initialization scripts by Gitlab.



Container Registry

Expiration policy will run in less than a minute

With the GitLab Container Registry, every project can have its own space to store images. More information

There are no container images stored for this project

With the Container Registry, every project can have its own space to store its Docker images. More Information

CLI Commands

If you are not already logged in, you need to authenticate to the Container Registry by using your GitLab username and password. If you have Two-Factor Authentication enabled, use a Personal Access Token instead of a password.

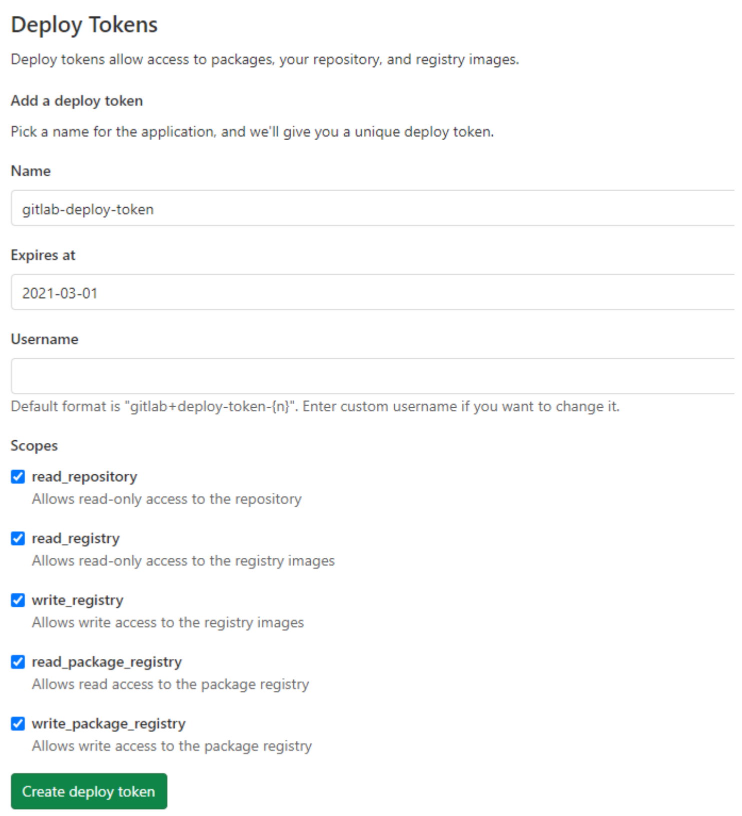
```
docker login registry.gitlab.com
```

You can add an image to this registry with the following commands:

```
docker build -t registry.gitlab.com/ds_20201/rea
```

Figure 16. GitLab Container Registry

In your repository, go to *Settings* → *Repository* → *Deploy token* and add the following token:



Deploy Tokens

Deploy tokens allow access to packages, your repository, and registry images.

Add a deploy token

Pick a name for the application, and we'll give you a unique deploy token.

Name: gitlab-deploy-token

Expires at: 2021-03-01

Username:

Default format is "gitlab+deploy-token-{n}". Enter custom username if you want to change it.

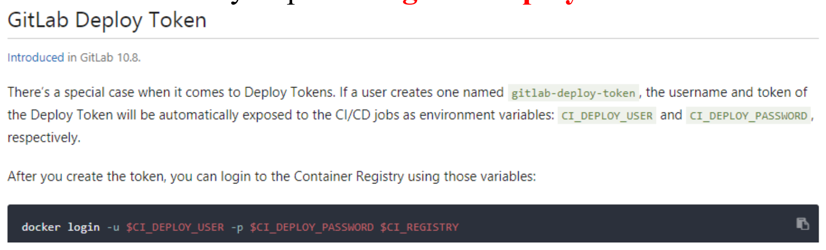
Scopes

- [x] read_repository — Allows read-only access to the repository
- [x] read_registry — Allows read-only access to the registry images
- [x] write_registry — Allows write access to the registry images
- [x] read_package_registry — Allows read access to the package registry
- [x] write_package_registry — Allows write access to the package registry

Figure 17. GitLab Deploy Token

Observations:

- !!! for the name field make sure you provide: **gitlab-deploy-token**



GitLab Deploy Token

Introduced in GitLab 10.8.

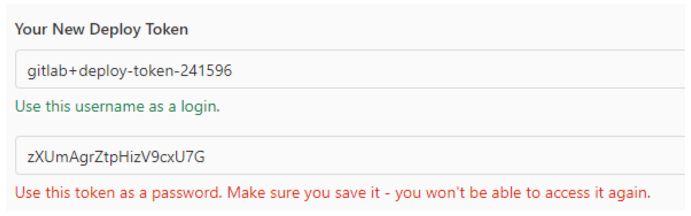
There's a special case when it comes to Deploy Tokens. If a user creates one named `gitlab-deploy-token`, the username and token of the Deploy Token will be automatically exposed to the CI/CD jobs as environment variables: `CI_DEPLOY_USER` and `CI_DEPLOY_PASSWORD`, respectively.

After you create the token, you can login to the Container Registry using those variables:

```
docker login -u $CI_DEPLOY_USER -p $CI_DEPLOY_PASSWORD $CI_REGISTRY
```

Figure 18 Gitlab Documentation [5]

- Check all the scopes as shown in Figure 17
- Make sure you save your credentials provided as a result. These credentials can be used to access the Gitlab registry locally from your computer as well.



Your New Deploy Token

gitlab+deploy-token-241596

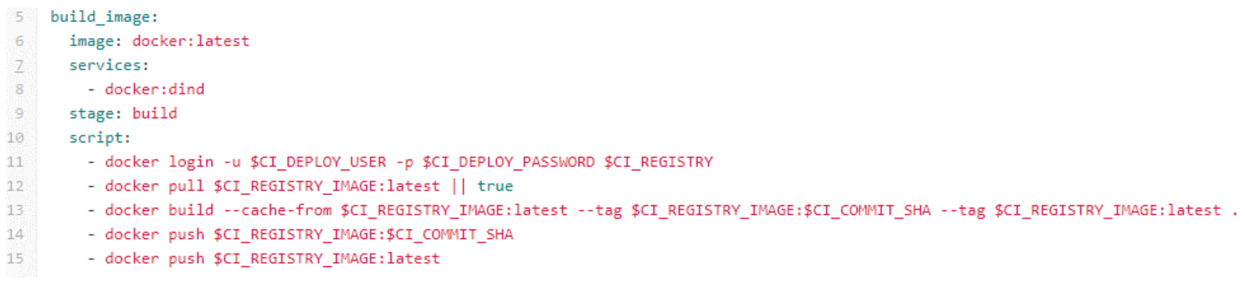
Use this username as a login.

zXUmAgrZtpHizV9cxU7G

Use this token as a password. Make sure you save it - you won't be able to access it again.

Figure 19. Credentials to access GitLab Docker Registry example

On your ***docker_production*** branch modify the .gitlab-ci.yml build phase to :



```
5  build_image:
6    image: docker:latest
7    services:
8      - docker:dind
9    stage: build
10   script:
11     - docker login -u $CI_DEPLOY_USER -p $CI_DEPLOY_PASSWORD $CI_REGISTRY
12     - docker pull $CI_REGISTRY_IMAGE:latest || true
13     - docker build --cache-from $CI_REGISTRY_IMAGE:latest --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA --tag $CI_REGISTRY_IMAGE:latest .
14     - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
15     - docker push $CI_REGISTRY_IMAGE:latest
```

Figure 20 Build phase using Docker

During the build stage presented in Figure 20:

1. Firstly, the authentication step will be run (line 11) and will use the credentials associated automatically to the ***gitlab-deploy-token*** previously created, to login to the Gitlab Registry.
2. At line 12 the latest imaged that has been build is pulled from the registry (considering the name of the image provided for the Variable CI_REGISTRY_IMAGE).
3. The pulled image (if any) will be considered as cache resource for the current build (line 13) and the current build is tagged with latest and tagged with the COMMIT_SHA.
4. The latest image is pushed in the Gitlab registry considering the 2 tags.

4.1.2. Test your solution by pushing an image to GitLab registry

- **Comment the deploy stage** from the .gitlab-ci. It will be updated accordingly in Section 4.2.
- Make a change in the code, commit it, and push it on your ***docker_production*** branch in order to trigger and test your build phase.
- On success, go to *Settings* → *Packages and Registries* → *Container Registry* and your image should be registered in the Image Repository with the two tags.

4.2. GitLab CD Automatic Deploy Docker Image on Heroku

The Heroku deployment can be configured and run from both the local computer and from Gitlab. In order to avoid configuration problems, **we recommend firstly deploying the application on Heroku from your local computer (section 4.2.1.), and when successful continue to setup the Gitlab CD stage (section 4.2.2.)**.
Regarding the docker files, keep in mind that Heroku will automatically launch the image created based on your Dockerfile, however the docker-compose file is NOT considered by Heroku, but it serves for your local tests. **Thus, please do not rely on any configuration inside the docker-compose file when deploying on Heroku**.

4.2.1. Configure Heroku locally

1. Install Heroku CLI on your computer [4]
2. Make sure you have docker installed on your computer.
3. Check your ***application.properties*** file from your spring boot application, and make sure you have your server.port exposed and set to : `server.port=${PORT:8080}.` The PORT variable is set by the Heroku runtime and incoming requests are forwarded to your application on this port.
4. In your project's root directory run:

```
> heroku login
> heroku container:login
> heroku config:set JAVA_TOOL_OPTIONS="-Xmx512m" --app spring-demo-ds2020
> heroku container:push  --app spring-demo-ds2020 web
> heroku container:release --app spring-demo-ds2020 web
> heroku logs --tail --app spring-demo-ds2020
```

Observation: make sure you use the application name set by you in the previous tutorial on heroku.

** For UBUNTU: In case of error regarding the login credentials on heroku container:login consider running the following cmds: sudo apt install gnupg2 pass* and *sudo docker login*

4.2.1.1. Test your solution locally

- Verify the logs of the application deployment using:
  ```
  > heroku logs --tail --app spring-demo-ds2020
  ```
- If successful, go to the Heroku Application page and open the Spring Boot application and check if it is up and running and retrieves data.
- If everything works ok, you can proceed and make the deployment step automatically using Gitlab CD.

4.2.2. Configure Heroku on Gitlab

In your repository, go to *Settings* → *CI/CD* → *Variables* and add the following 2 variables:

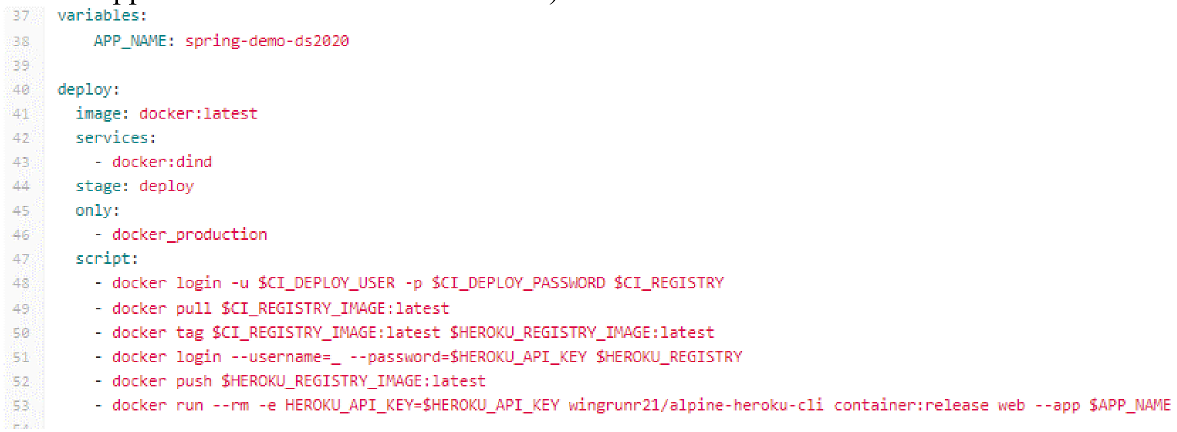
- **Key**: HEROKU_REGISTRY
  Value: registry.heroku.com
- **Key**: HEROKU_REGISTRY_IMAGE
  Value: registry.heroku.com/spring-demo-ds2020/web

Observation: make sure you use the application name set by you in the previous tutorial on heroku.

Add the deploy stage to your .gitlab-ci file, making sure that you set the restrictions for only the current branch, the **docker_production** branch.

During the deploy stage presented in Figure 21:

1. Firstly, the authentication Gitlab step will be run (line 48) to connect to the Gitlab registry
2. At line 49 the latest imaged that has been build is pulled from the registry (considering the name of the image provided for the Variable CI_REGISTRY_IMAGE).
3. The pulled image is tagged with the name required by the Heroku Registry (line 50).
4. The Heroku authentication step will be run (line 51) that will use the HEROKU_API_KEY set in the previous tutorial in Gitlab, and connect to the Heroku registry
5. The tagged image is pushed in the Heroku registry considering (line 52).
6. The imaged is released -line 53 (launched on the Heroku platform corresponding to the application set on Heroku – line 38)



```
37 variables:
38     APP_NAME: spring-demo-ds2020
39
40 deploy:
41   image: docker:latest
42   services:
43     - docker:dind
44   stage: deploy
45   only:
46     - docker_production
47   script:
48     - docker login -u $CI_DEPLOY_USER -p $CI_DEPLOY_PASSWORD $CI_REGISTRY
49     - docker pull $CI_REGISTRY_IMAGE:latest
50     - docker tag $CI_REGISTRY_IMAGE:latest $HEROKU_REGISTRY_IMAGE:latest
51     - docker login --username=_ --password=$HEROKU_API_KEY $HEROKU_REGISTRY
52     - docker push $HEROKU_REGISTRY_IMAGE:latest
53     - docker run --rm -e HEROKU_API_KEY=$HEROKU_API_KEY wingrunr21/alpine-heroku-cli container:release web --app $APP_NAME
```

Figure 21 Deploy Phase using Docker

4.2.2.1. Test your solution on Gitlab

- Make a modification in your code, commit it and push it to the **docker_production** branch in order to trigger the pipeline and the deploy stage
- Check if the pipeline runs successful and on success validate that the application is successfully launched by Heroku.
- If the application launch fails, you can verify the logs of the application deployment on your local computer using:

  > heroku logs --tail --app spring-demo-ds2020

5. Further development

- Apply the configuration from section 4 to deploy the React Application on Heroku.
- Modify the host.js file by referencing the public IP address of the Spring App container.
- Deploy the application resulted from Assignment 1 on Heroku cloud.

References

[1] https://gitlab.com/ds_20201/spring-demo
[2] https://www.baeldung.com/spring-boot-docker-images
[3] https://gitlab.com/ds_20201/react-demo
[4] https://devcenter.heroku.com/articles/heroku-cli#download-and-install
[5] https://docs.gitlab.com/ee/user/project/deploy_tokens/
[6] https://developer.okta.com/blog/2020/06/24/heroku-docker-react
[7] https://docs.docker.com/install/linux/docker-ce/ubuntu/
[8] https://www.computerhope.com/jargon/u/user-space.htm
[9] https://en.wikipedia.org/wiki/Hypervisor
[10] https://devopscon.io/blog/docker/docker-vs-virtual-machine-where-are-the-differences/
[11] https://www.docker.com/blog/vm-or-containers/
[12] https://www.sciencedirect.com/topics/computer-science/hypervisors
[13] https://docs.docker.com/registry/
[14] https://docs.docker.com/develop/develop-images/baseimages/
[15] https://en.wikipedia.org/wiki/Virtual_machine