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 Azure

Rancea Alexandru Marcel Antal Claudia Antal

2023-2024

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.

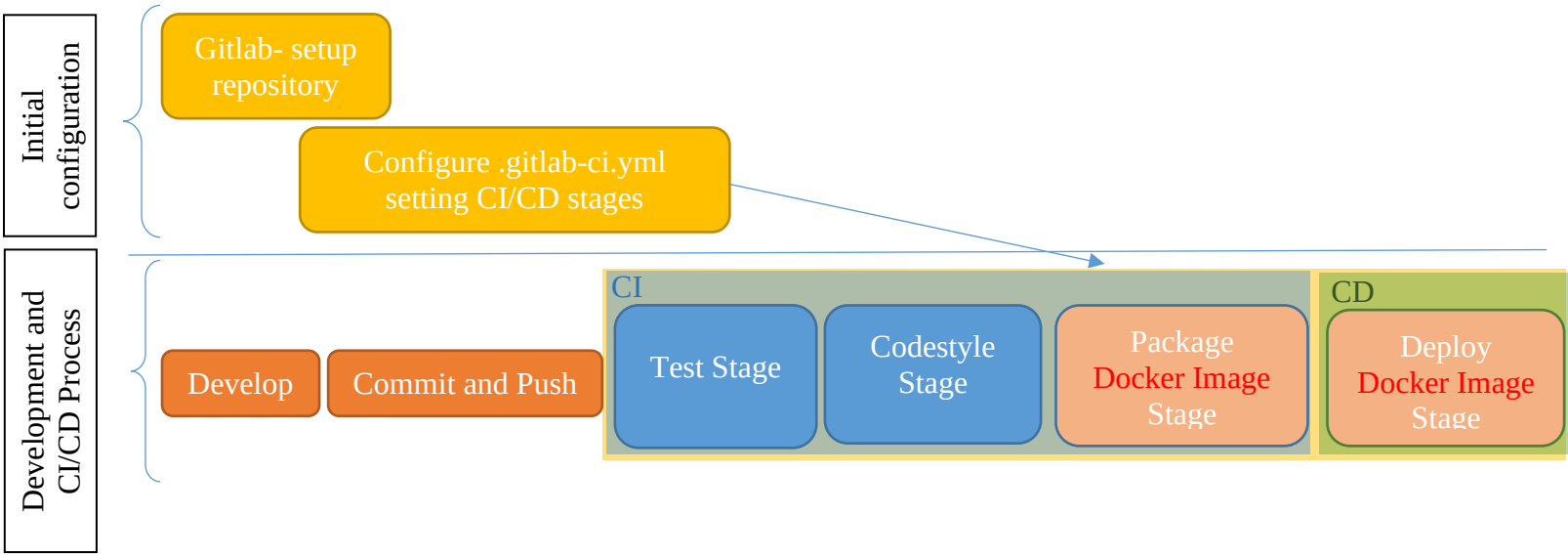

Figure 1. CI/CD Pipeline

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.

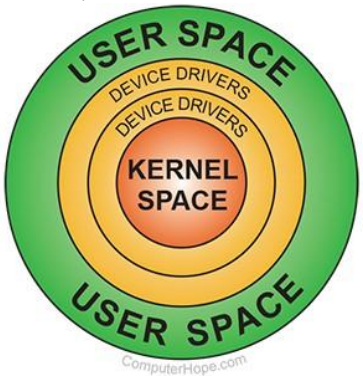

Figure 2. Computer Memory privileges separation (Source [8]).

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

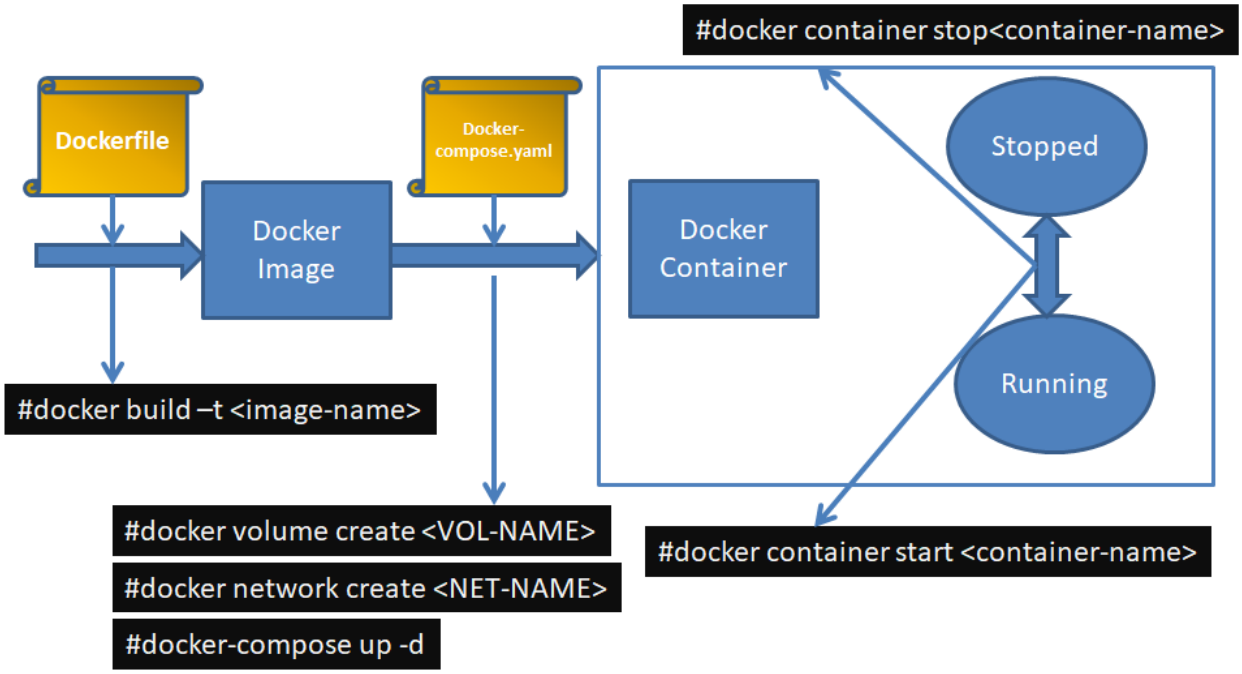

Figure 3. Docker lifecycle and basic commands

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).

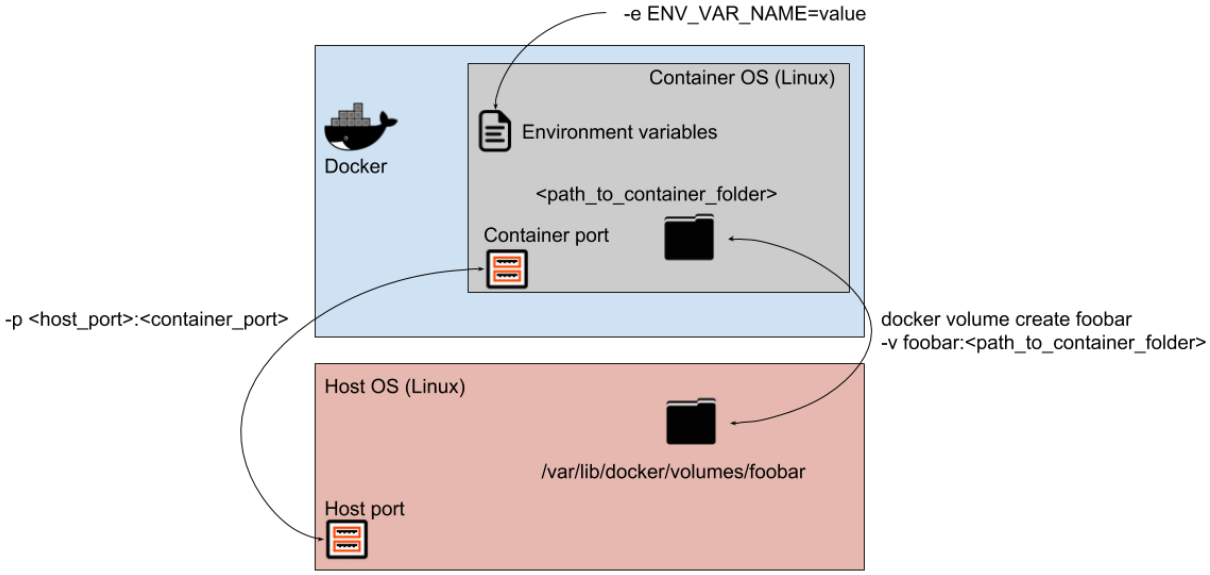

Figure 4. A docker container and communication with the host (port mapping and shared folders using volumes)

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. (aici trb sa discutam)

5. Lines 14-19 from the Dockerfile presented in Figure 6 contain the DB credentials from the PostgreSQL that is found locally either by installing Postgres or by using the latest Postgres Docker Image (docker pull postgres)

Dockerfile 1022 Bytes Edit Web IDE

```
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

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 7 lines 15-19) so the specified values are considered, and the default ones are ignored.

```
1  ##############################################
2  ### DATABASE CONNECTIVITY CONFIGURATIONS ###
3  ##############################################
4  database.ip = ${DB_IP:localhost}
5  database.port = ${DB_PORT:5432}
6  database.user = ${DB_USER:root}
7  database.password = ${DB_PASSWORD:root}
8  database.name = ${DB_DBNAME:city-db}
```

Figure 7. 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")

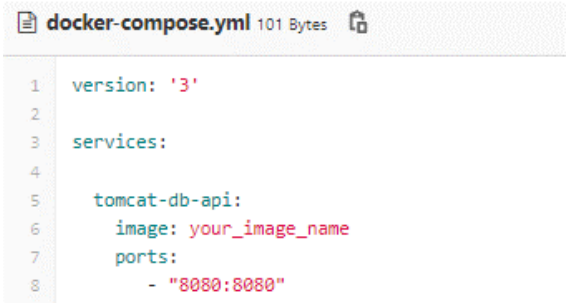

Figure 8. Docker-compose file of spring-demo image

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:

Dockerfile 573 Bytes Edit Web IDE

```
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 9. 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 10. 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

```yaml
version: '3'

services:

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

Figure 11. 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.

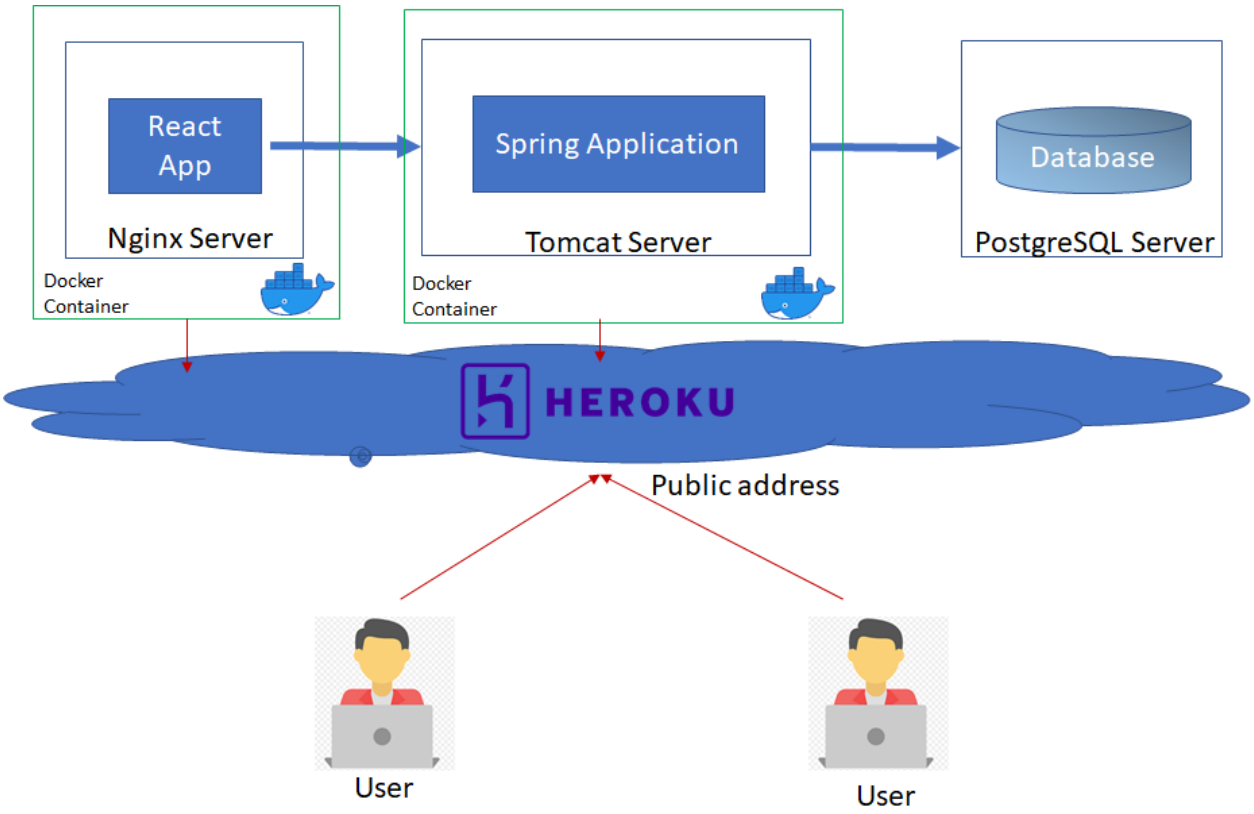

Figure 12. Project Deployment Diagram

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.

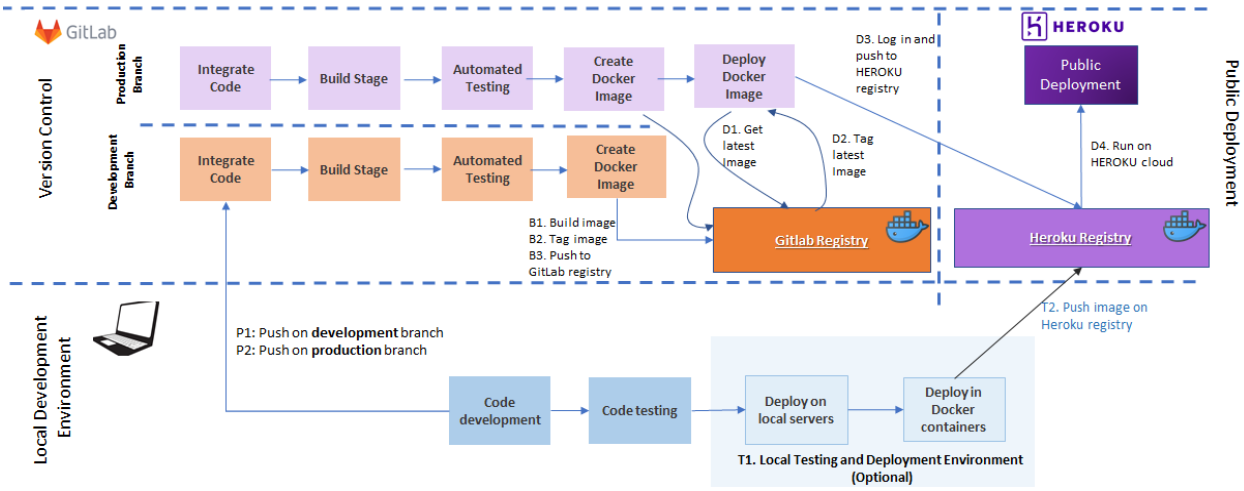

Figure 13. Project Deployment Pipeline

The steps that are involved in deploying the application on the Heroku cloud are depicted in Figure 13 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 Azure Container Repository. Push the image to Azure Container Repository.	Section 4.2.1.
B1	Build the image from the code pushed and built on one of the branches from GitHub.	Section 4.1.1.
B2	Tag the built image	
B3	Push the image on Azure Container Repository	
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. Azure CI/CD using Docker

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

- The first part of this section covers the setup for resource groups and the container registry.
- The second part of this section covers the setup to automate the process of creating images by the CI pipeline from GitHub and save them in Azure Container Repository.
- The third part of this section covers the setup to automate the process of deployment by the CD pipeline from Azure.

4.1. Azure CI Automatic Build Docker Image

Consider the following setup on your ***docker_production*** branch. Set your branch as protected by going on GitHub to *Settings* → *Branches* → *Add rule.*

4.1.1. Configure Azure

Visit https://azureforeducation.microsoft.com/devtools → *Sign in* and use your student email (@student.utcluj.ro) to complete the account creation.

Press the *Claim your Azure credit now* and complete with your own personal information.

After the log in, select from the top bar, the console/terminal icon. Select your preferred command line interpreter, and press Create storage.

To create a resource group, run the following command in the Azure terminal, created one step before: az group create --name nameLastNameGroupNumber --location germanywestcentral and change the name with your own information.

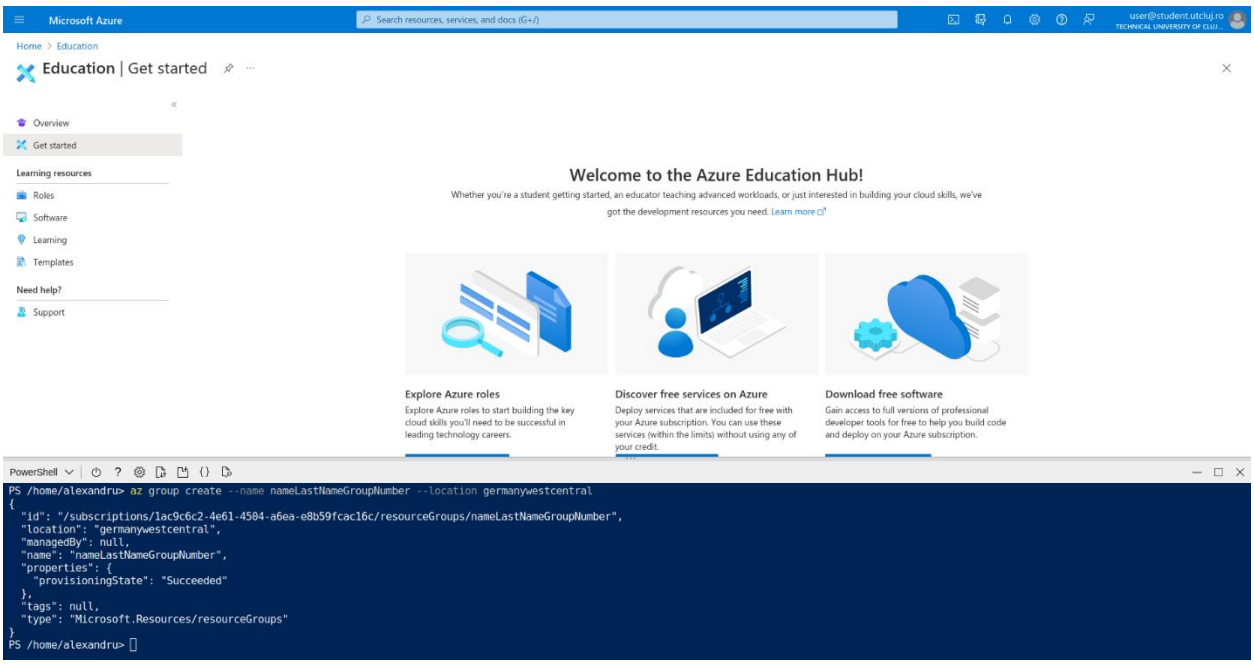

Figure 14. Azure Resource Group

After the resource group has been created, create a container registry by the following command: az acr create --resource-group nameLastNameGroupNumber --name containerregistrynamelastnamegroupnumber --sku Basic and replace the --resource-group name with the one created above and --name with your own information (the container registry name should not contain capital letters).

Log out and log in again, if the Azure terminal says that there is no available subscription.

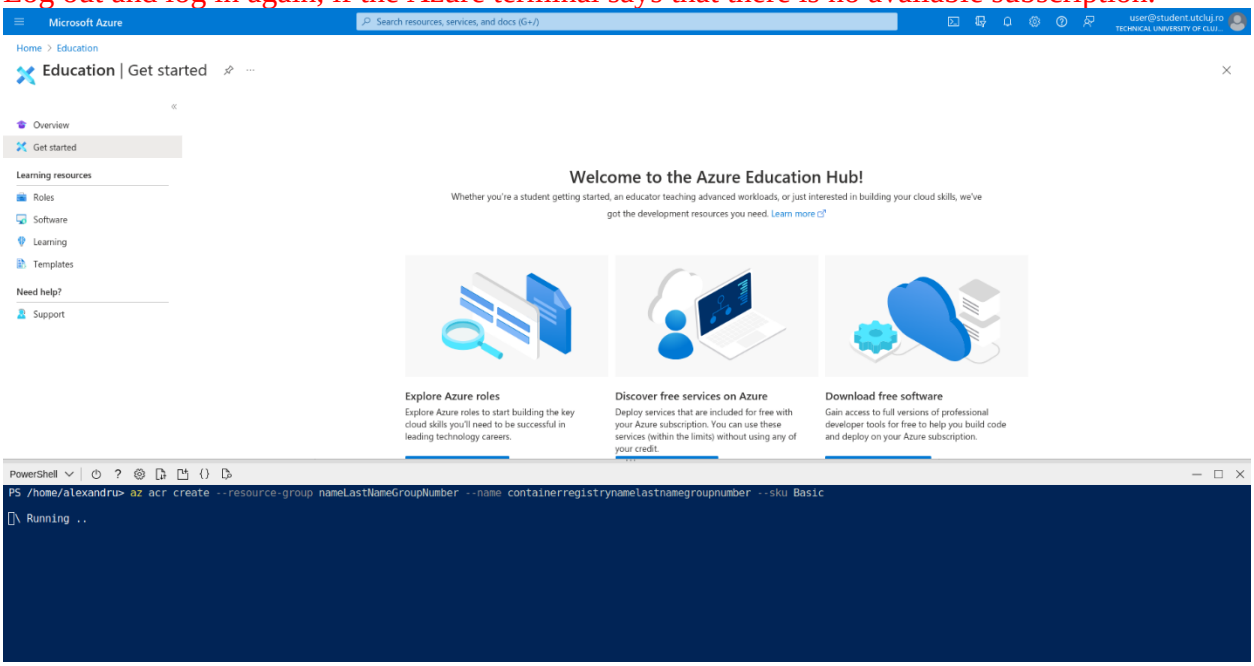

Figure 15. Azure Container Registry

To visualize the last two steps, select the *Menu bar* → *Resource groups* → *nameLastNameGroupNumber*. At this point, the only resource created in our group is the container registry.

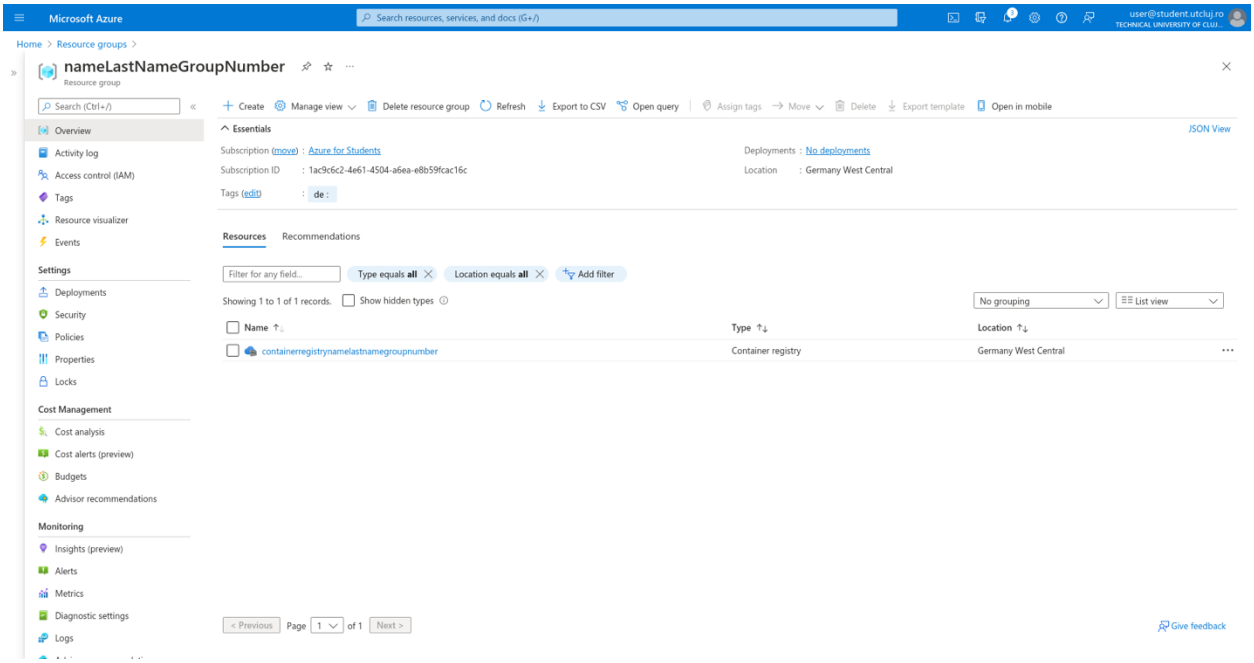

Figure 16. Azure Resource Group and Container Registry

A mandatory example to do before moving forward is to push to latest Postgres image in our Azure Container Repository. This is done with the following steps:

1. Open a CLI (Command Line Interpreter) locally (on your personal computer), and login into the Azure Container Registry:

```
docker login containerregistrynamelastnamegroupnumber.azurecr.io -u containerregistrynamelastnamegroupnumber –p sTsnMgSQa0dx5LD=qWqEtcIv6zmJSmMP;
```

If you receive the following error: accepts at most 1 arg(s), received 3, this means you are on a Windows machine and need to run the following command:

```
docker login -u containerregistrynamelastnamegroupnumber –p sTsnMgSQa0dx5LD=qWqEtcIv6zmJSmMP containerregistrynamelastnamegroupnumber.azurecr.io;
```

The credentials can be found in: *Home* → *Resource Group* → *nameLastNameGroupNumber* → *containerregistrynamegroupnumber* → *Access keys* and by enabling the Admin user. Replace the Login server, Username and password with your own details. Use the first password.

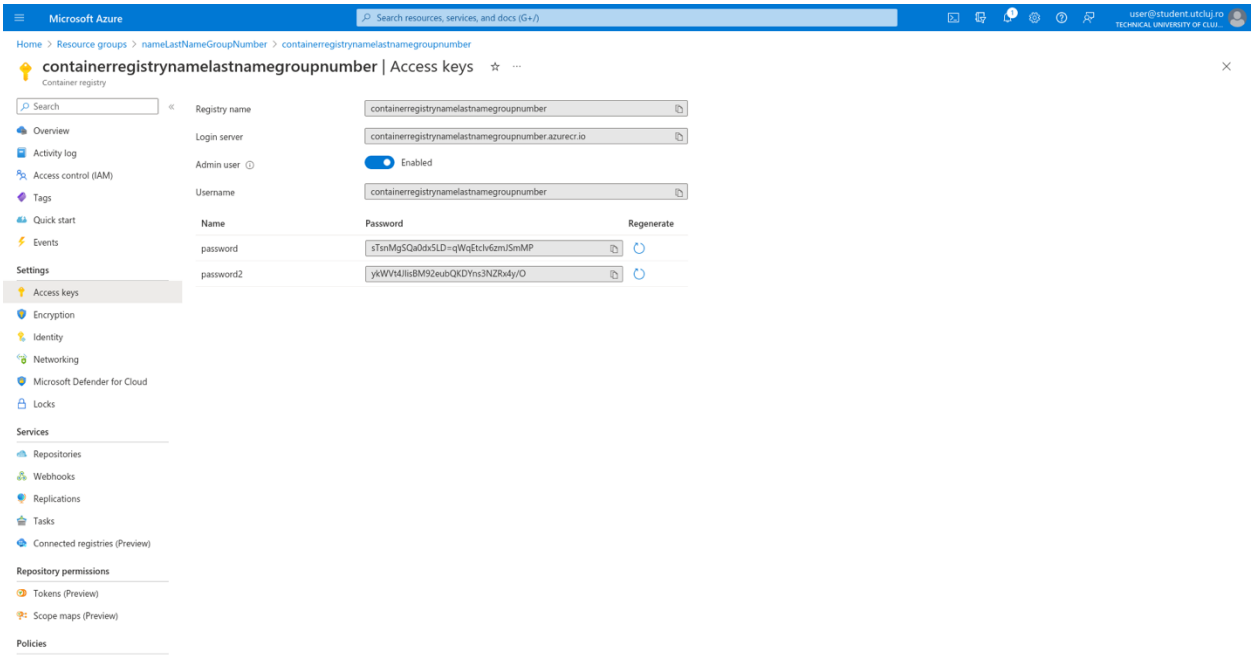

Figure 17. Azure Container Registry Credentials

2. Pull the latest docker image for Postgres

```
docker pull postgres;
```

3. Create a tag for the Postgres image

```
docker tag postgres containerregistrynamelastnamegroupnumber.azurecr.io/db
```

Modify the containerregistrynamelastnamegroupnumber with your details.

4. Push the image to your Azure Container Registry

```
docker push containerregistrynamelastnamegroupnumber.azurecr.io/db
```

Modify the containerregistrynamelastnamegroupnumber with your details.
5. Go to Home->Resource Groups → nameLastNameGroupNumber → containerregistrynamegroupnumber → Repositories to check if the image is in your Azure Container Repository.

4.1.2 Azure DevOps

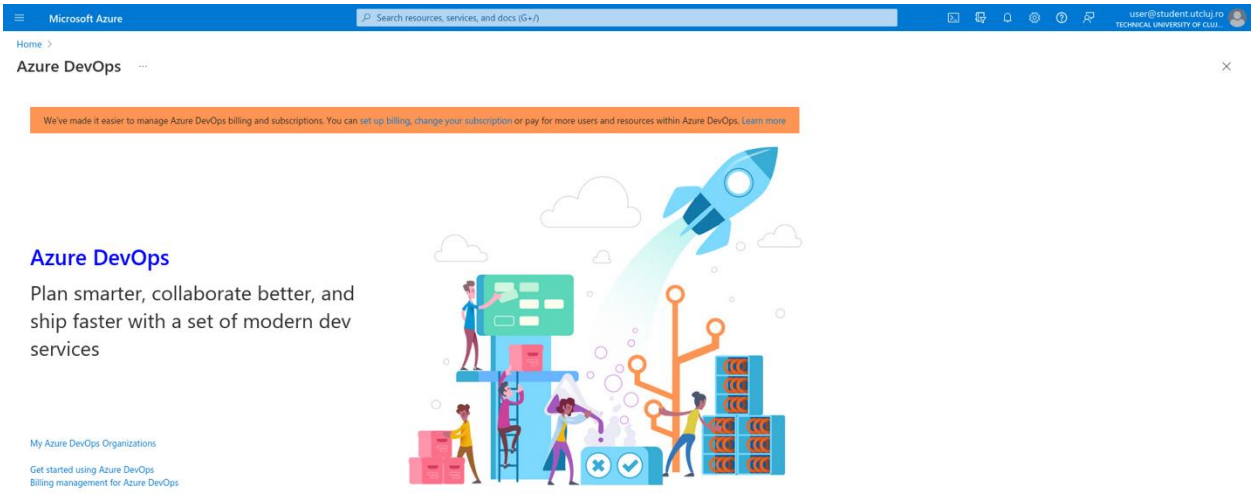

Figure 18. Azure DevOps

Before developing the CI/CD pipeline, go to *My Azure DevOps Organziations* → *Create new organization* and follow the steps presented in Figure 19 and Figure 20.

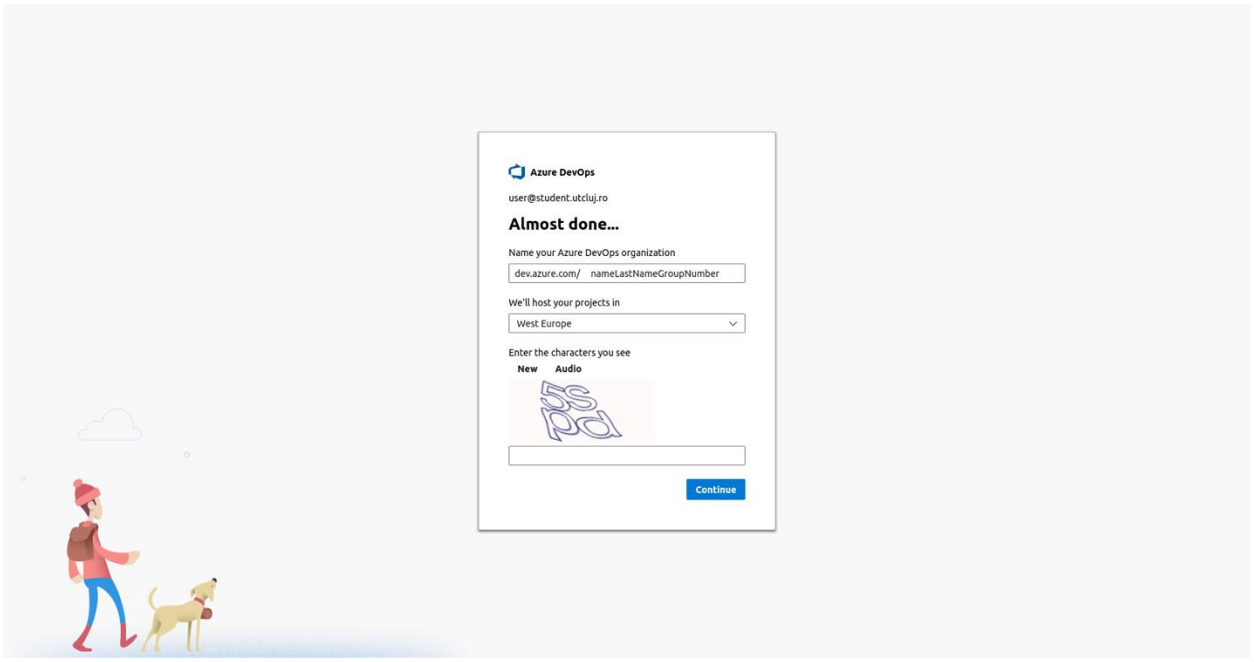

Figure 19. Create Azure DevOps Organization

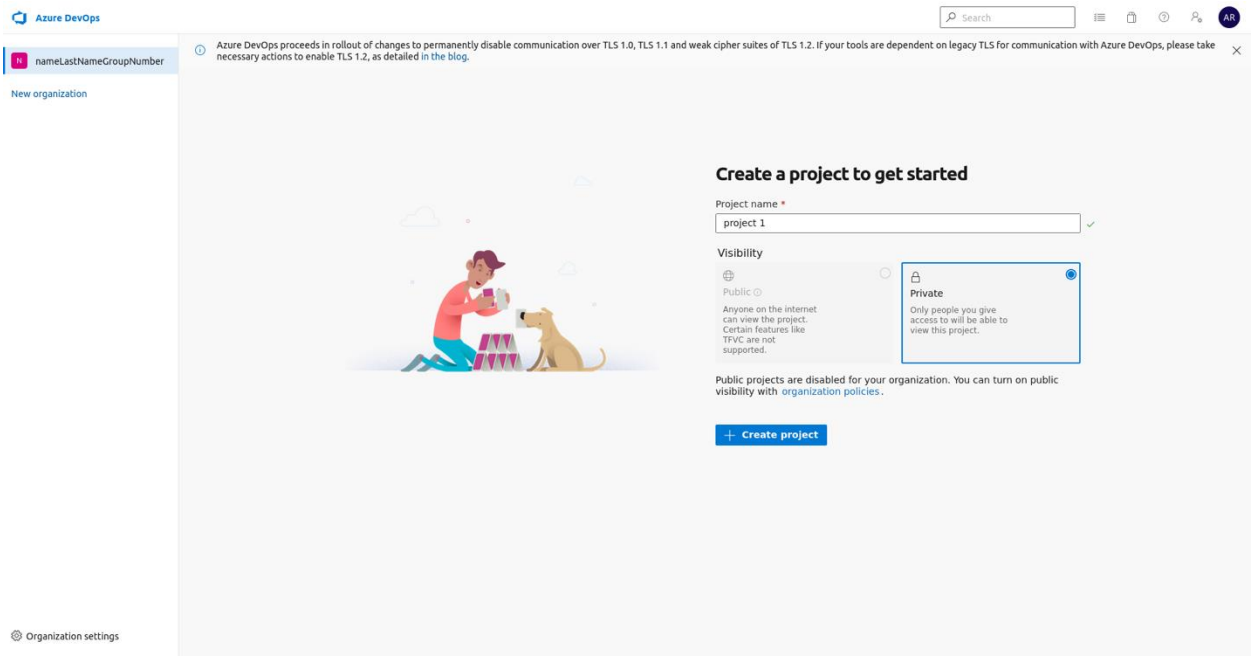

Figure 20. Create a project in Azure DevOps

4.1.3 Azure Pipeline Agent

To be able to run the CI/CD pipeline, we need an Agent that is responsible for all the instructions. Due to student account limitations, we must run the Agent locally. Select *Project settings* from the bottom-left of the screen → *Agent pools* → *Add pool* and select Self-hosted for Pool type and name it local.

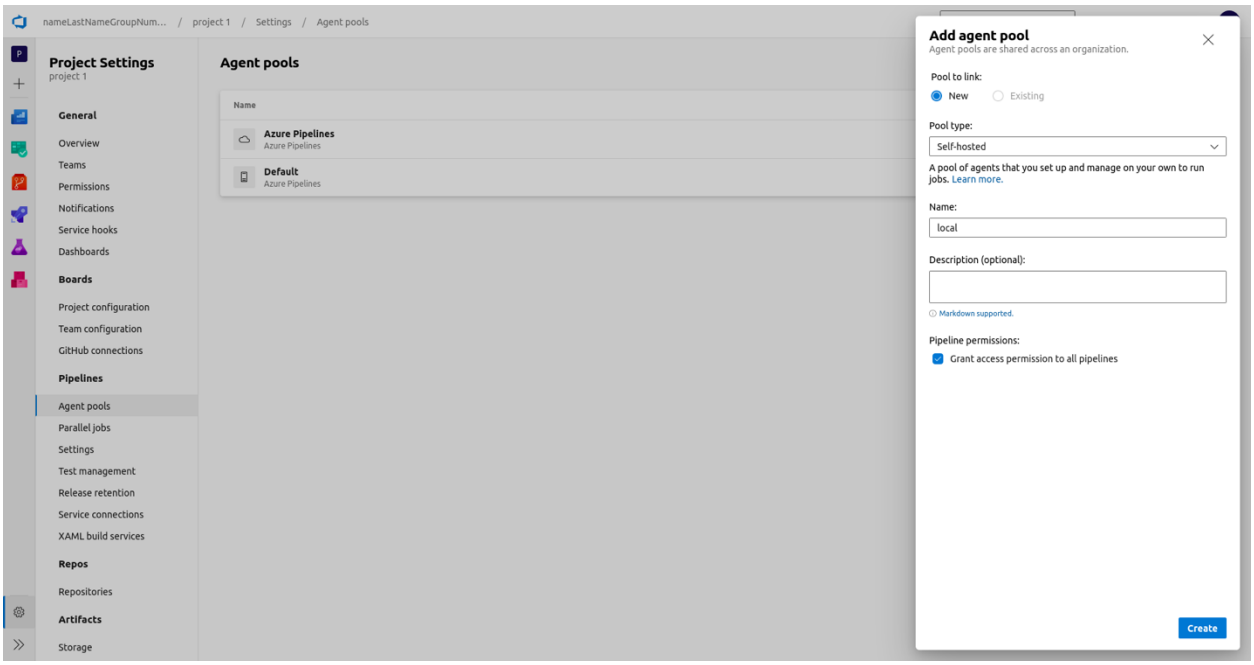

Figure 21. Agent pool

The list of agent pools should refresh, and a new item named local should appear. The Azure Pipeline Agent is used to:

1. Build the image for the code from GitHub, push it on Docker Registry
2. Get the image from Docker Registry and deploy the container on Docker on Azure

Download the Ubuntu VM Image linked https://drive.google.com/drive/folders/1A5_L8WHCgsUutVbfwtl7ZMQg4uPPufjY?usp=sharing
Install Virtual Box (https://www.virtualbox.org/wiki/Downloads) and open the VM Image downloaded. It has the Azure Pipeline Agent preinstalled and all you have to do is to configure the access credentials.

Note:
If you use the agent from the VM, login intro the VM and use the Ubuntu commands listed below.
If you use a Windows machine, use the corresponding commands.

Before running the config command, we need an access token. Select the *icon highlighted* in Figure 22 → *Personal access tokens* → *New Token* and configure the token as presented in Figure 23. Note: you can increase the expiration date to your needs.

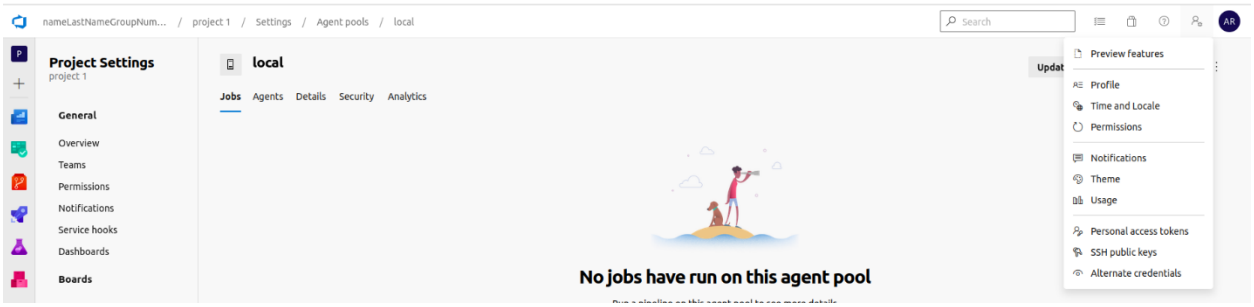

*Figure 22. Personal Access Tokens *trebe highlight la imgt**

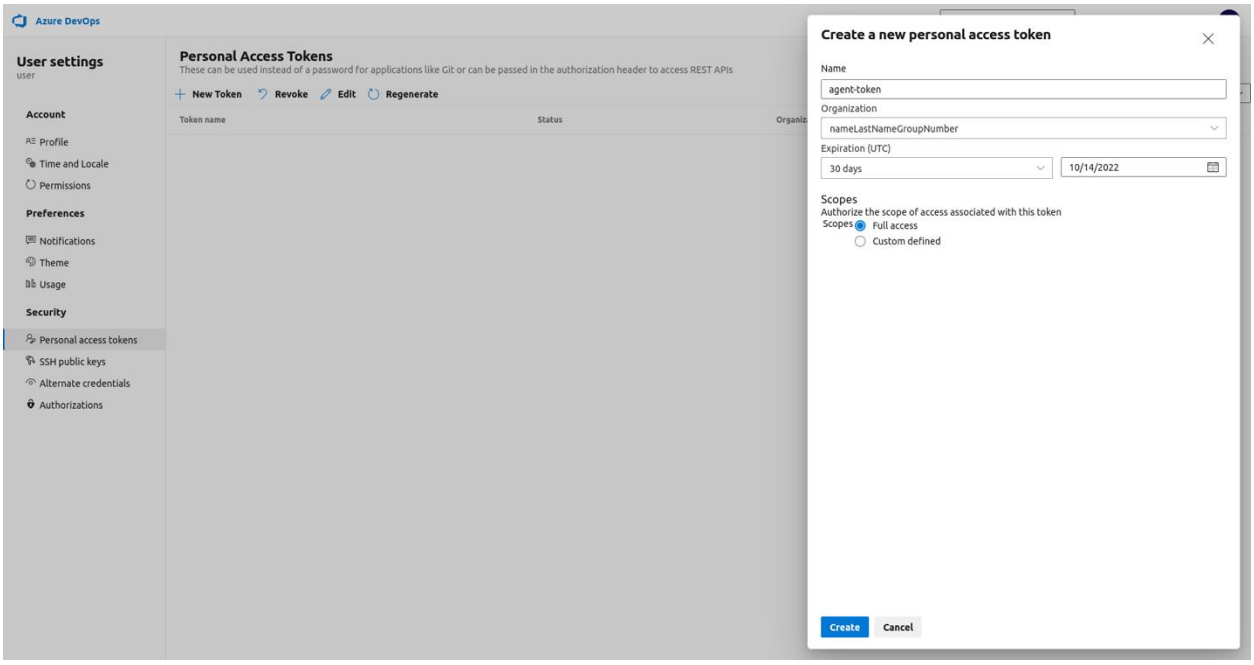

Figure 23. Personal Access Token Settings

Make sure you are saving the token displayed on screen, locally on a text file. If you forget the personal token, you need to regenerate the token or create a new one and change the value everywhere it was used.

Login into the VM and using your favorite command line interpreter (Bash/PowerShell), you must be within the agent's directory before trying to run the config command - Ubuntu, .\config.cmd - Windows).

The configuration is the following:

1. your organization server url → https://dev.azure.com/nameLastNameGroupNumber
2. enter authentication type (press enter for PAT) - press enter
3. personal access token → the personal access token generated in Figure 23
4. agent pool → local
5. agent name → agent
6. press enter

If you use a Windows machine only enter Y for run agent as Service. Y again, Enter, Enter

Figure 24. Agent Configuration

Prerequisites

1. Add user to docker group:
   sudo usermod -aG docker $USER
   newgrp docker
2. Login Azure
   docker login azure
   Login with your azure credentials
   Verify if *dockerAccessToken.json exists in /home/$USER/.azure*

To start the agent, execute the .\run.cmd (Only if you haven't set the agent as a Service) - Windows or ./run.sh - Ubuntu command. To install the service (Ubuntu only), run the following commands: sudo ./svc.sh install and sudo ./svc.sh start. To visualize the agent's tasks run sudo ./svc.sh status.

Figure 25. Agent Status

To verify that all the configurations are right and the connection with the Azure DevOps platform is done, go to, *Project Settings* → *Agent pools* → *local* → *Agents*. Under the name of the agent you can see his status.

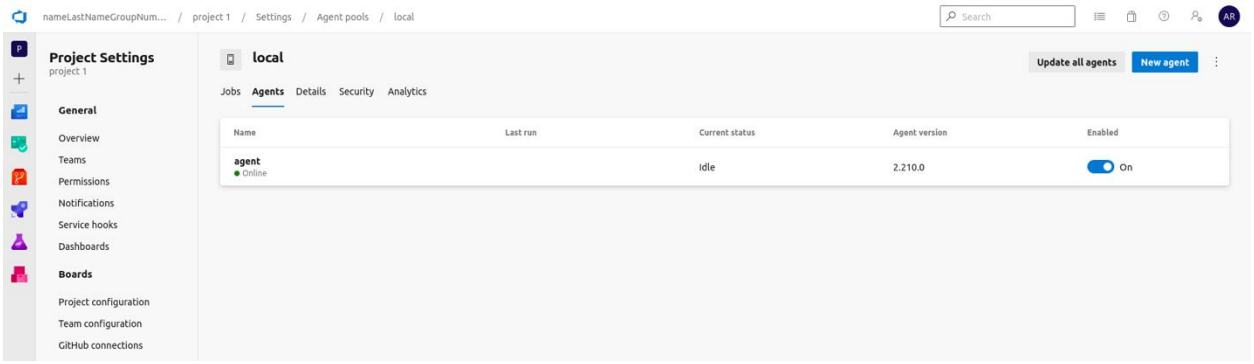

Figure 26. Local Agent Status Visualization

4.1.4 Azure CI

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

To create a pipeline, you must select *Pipelines* → *New pipeline*. Because our code is present in a GitHub repository, we will continue by selecting *GitHub*.

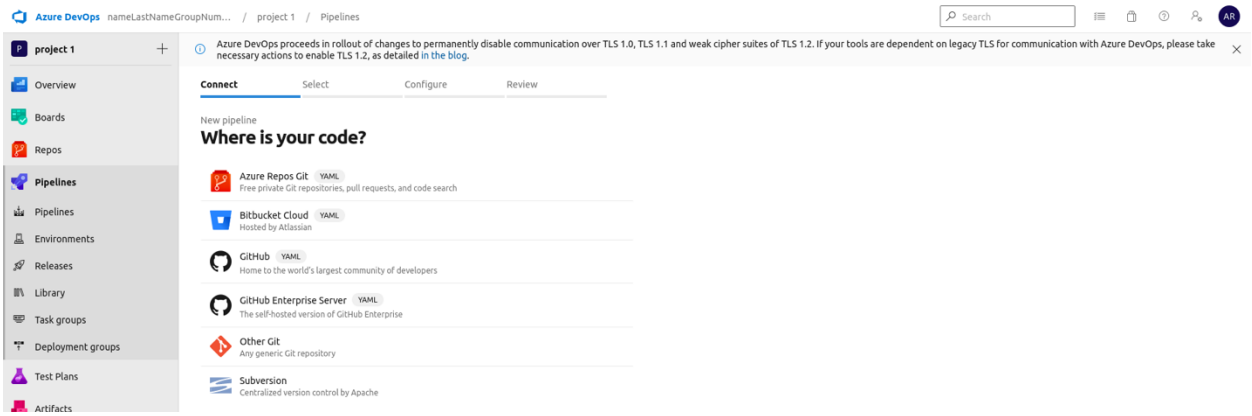

Figure 27. Pipeline Creation – GitHub

After you log in with your GitHub credentials, you should select your repository.

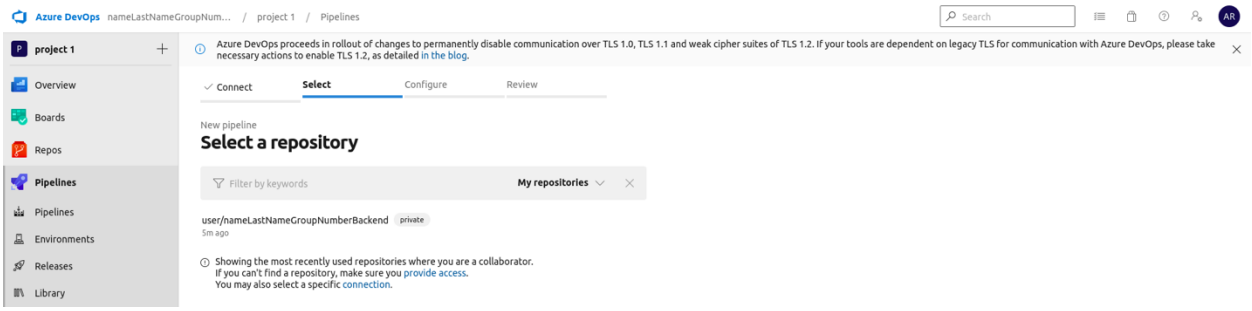

Figure 28. GitHub Available Repositories

You will be redirected to GitHub, scroll down to Repository access section, select Only selected repositories and press Approve and Install.

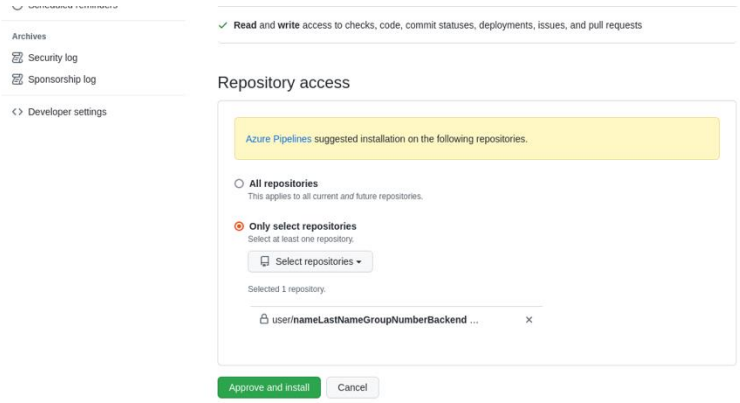

Figure 29. GitHub Repository Access Section

To configure the pipeline, select Docker (Build and push an image to Azure Container Registry) and configure the container registry name with the one you created in the first step of this tutorial.

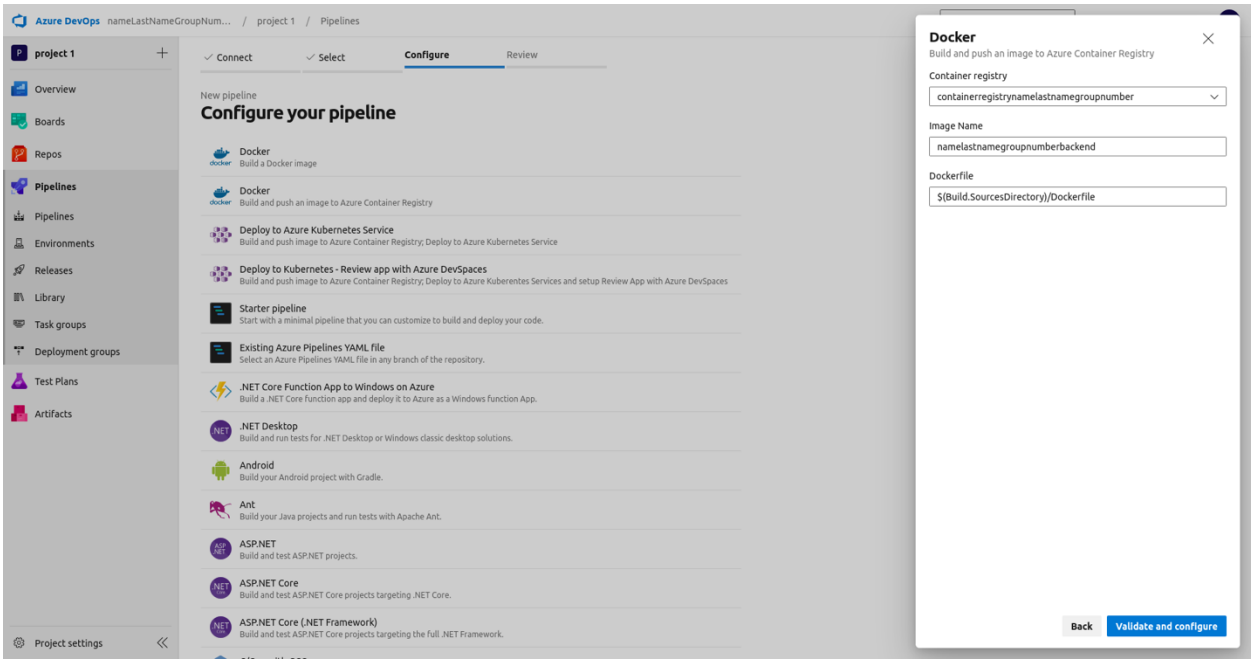

Figure 30. Configure your pipeline - Docker

You must replace the code after the stages tag with the X proprieties found on GitHub. Select save and run.

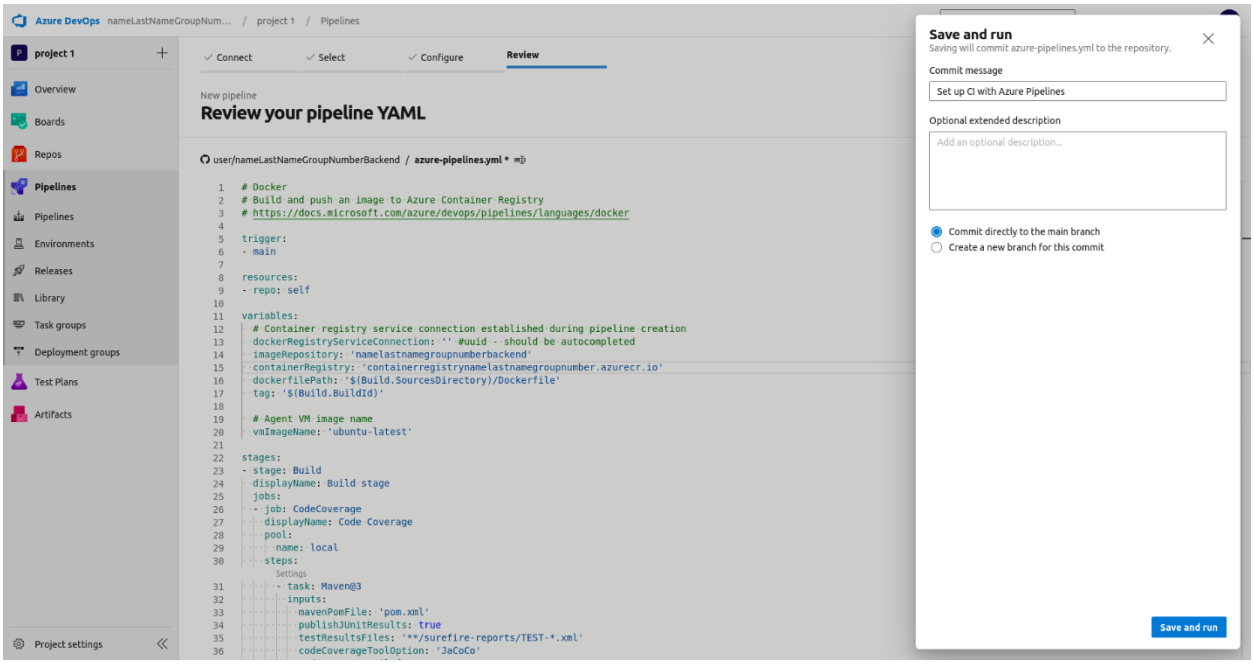

Figure 31. Pipeline Stages Setup

After the job is finished, there are 3 main tabs available. On the summary tab (Figure 32), we can verify that the project builds correctly. On the Tests tab (Figure 33) we can check that the tests run and how many are successful and on the Code Coverage tab, we can see the code style issues (Figure 34).

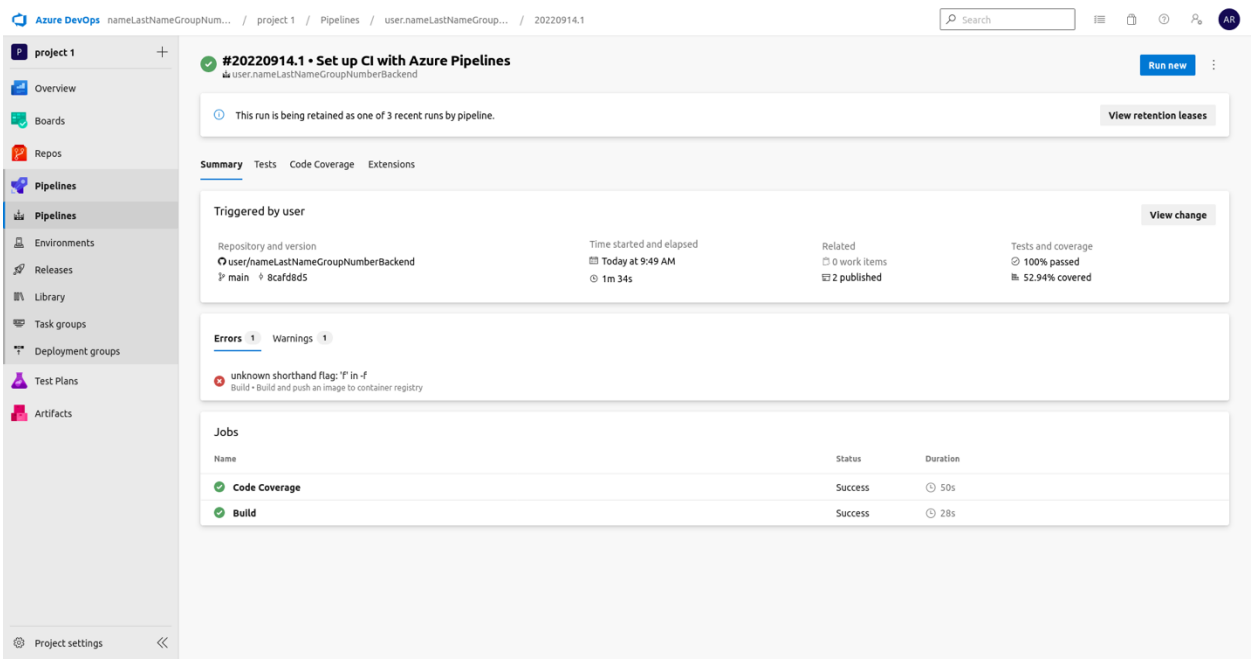

Figure 32. CI Pipeline Summary

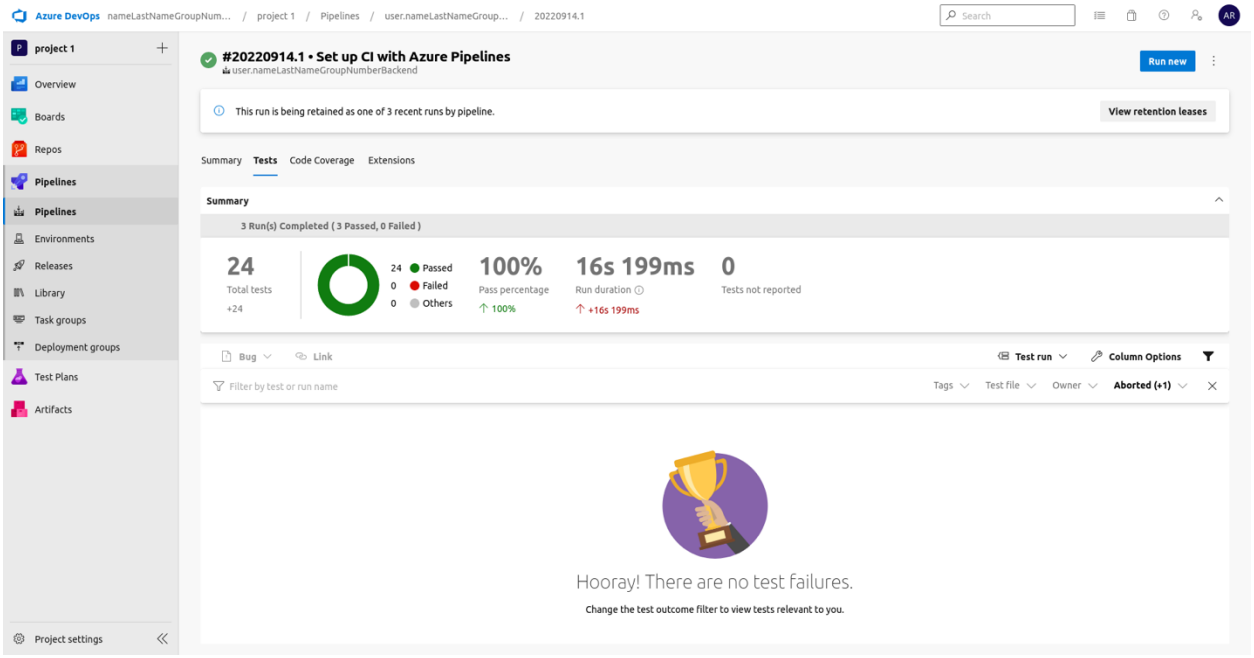

Figure 33. CI Pipeline Tests

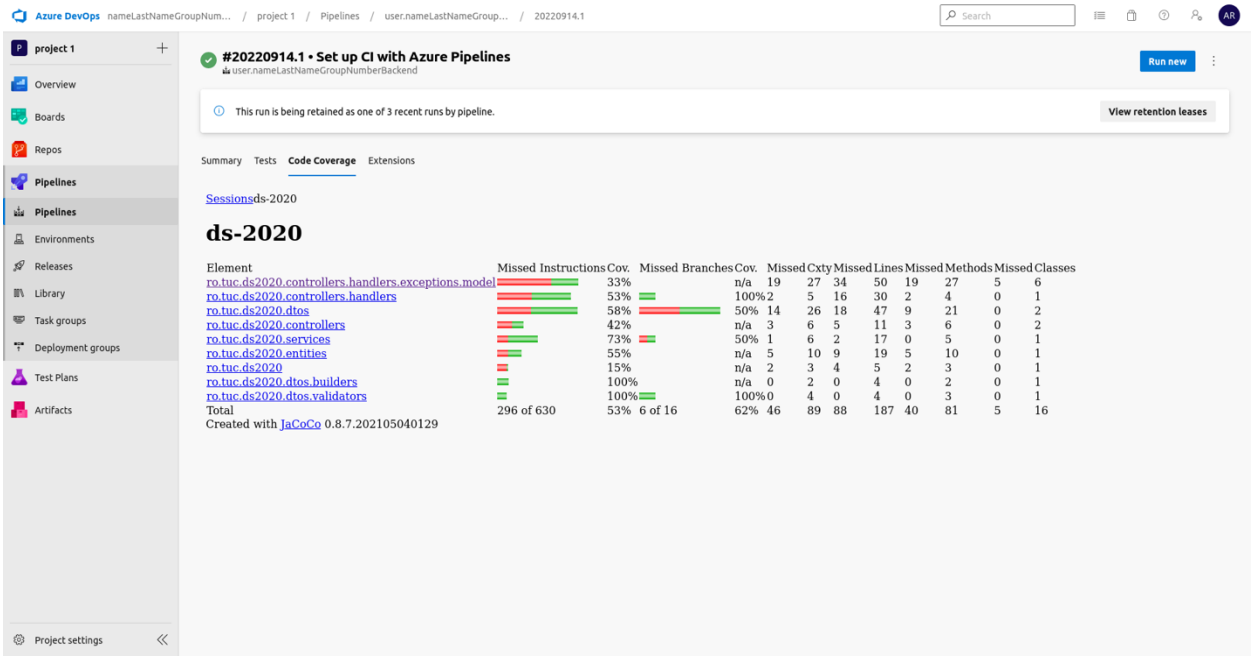

Figure 34. CI Pipeline Code Coverage

4.1.5 Azure CD

The continuous deployment aims at delivering as fast as possible the new features added through your code to the deployment servers. If all the previous steps (build, test, and check style) are successful, it will proceed with the deployment of your application on a server. 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.

To create a pipeline, you must select *Pipelines* → *Releases* → *New pipeline*. Change the stage name to Development.

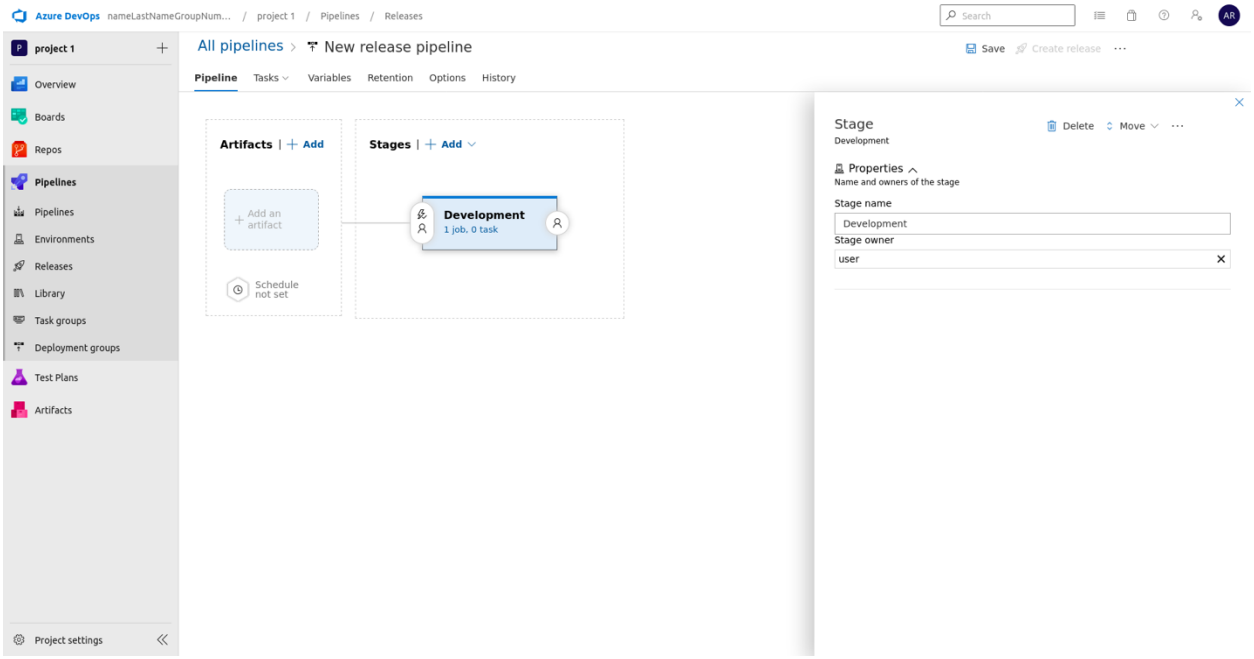

Figure 35. Create a CD Pipeline

Before starting with the tasks within the pipeline, a connection between the CI and CD must be set up. This is done by pressing on *Add an artifact* → select Source type as Build, select your current Azure DevOps project, the build pipeline and the default version.

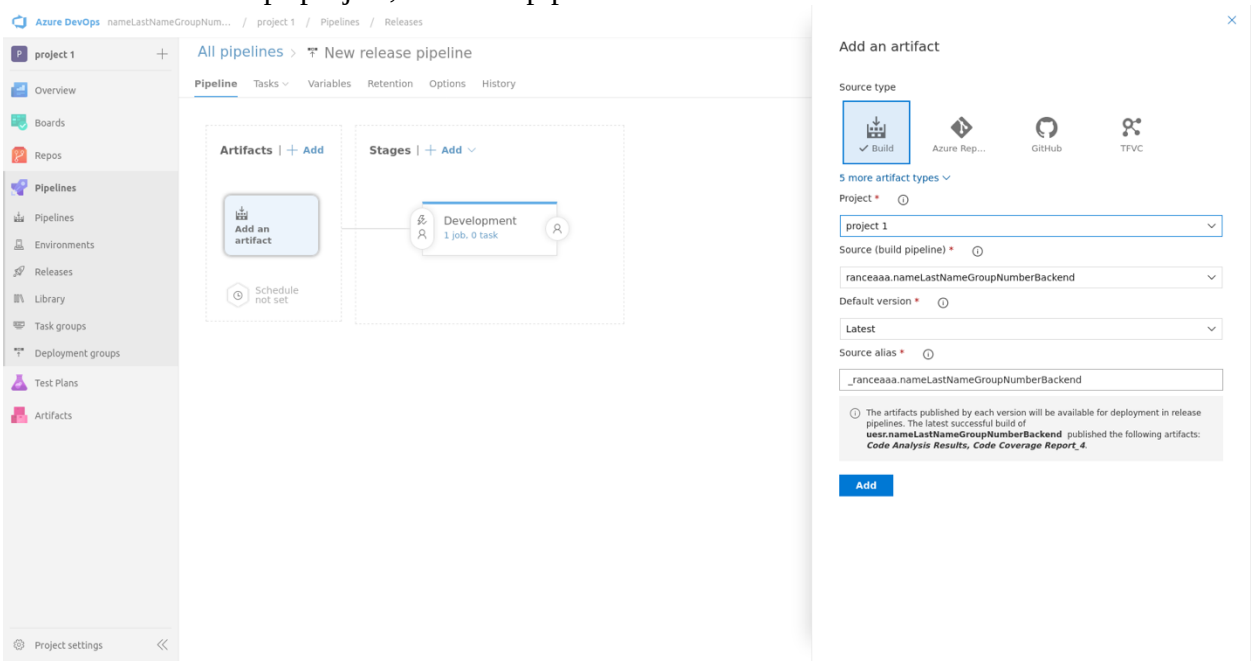

Figure 36. Add an Artifact - Build

After the artifact has been added, press the lightning icon to creates a release every time a new built is available.

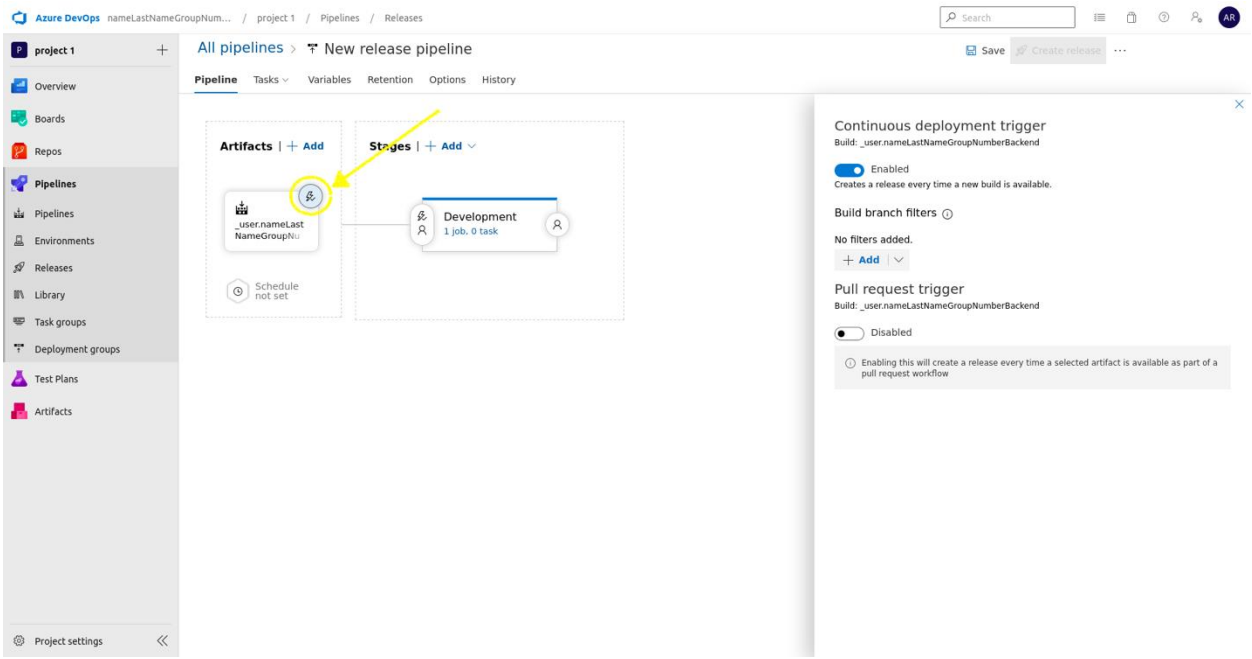

Figure 37. Trigger the CD when a Build was successfully added

Before the CD pipeline starts, the Docker Compose file needs to be available. We will do this by adding another artifact. Select Source type as GitHub, and select the Source (repository) to your repository where the Docker Compose file is found. Select the Default branch and the Default Version to Latest from the default branch.

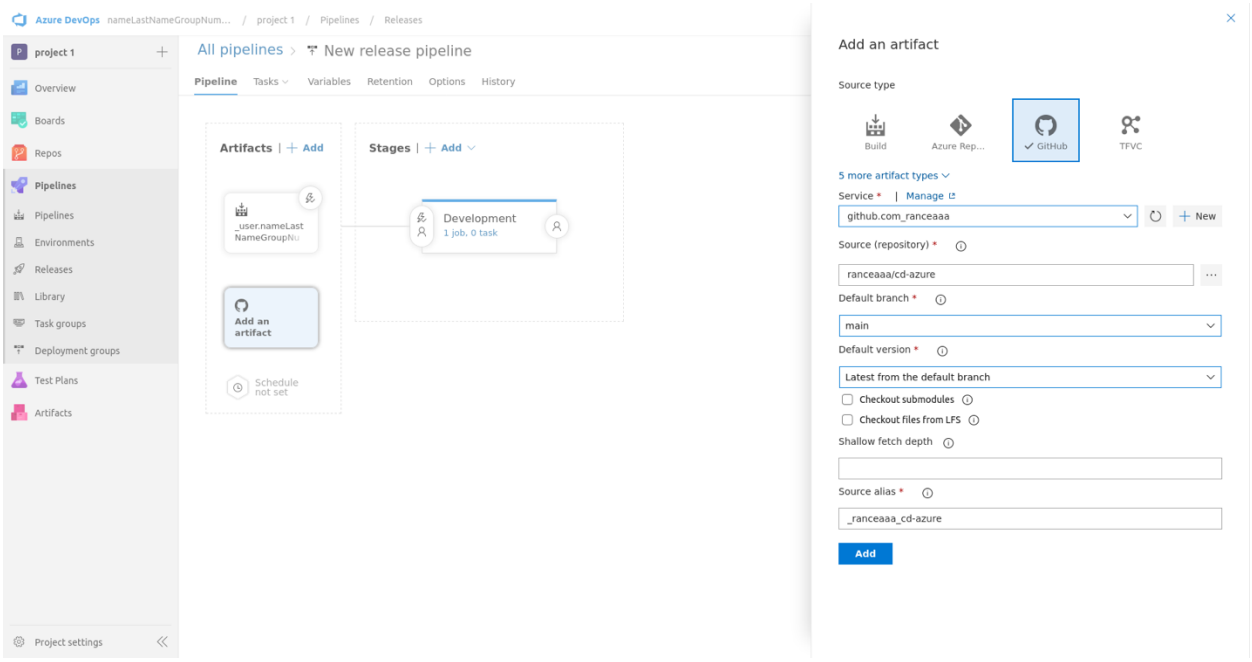

Figure 38. Add an Artifact with the Docker Compose file

To configure the pipeline to work with our local agent. *Tasks* → *Agent job* → *Agent pool* → *local.*

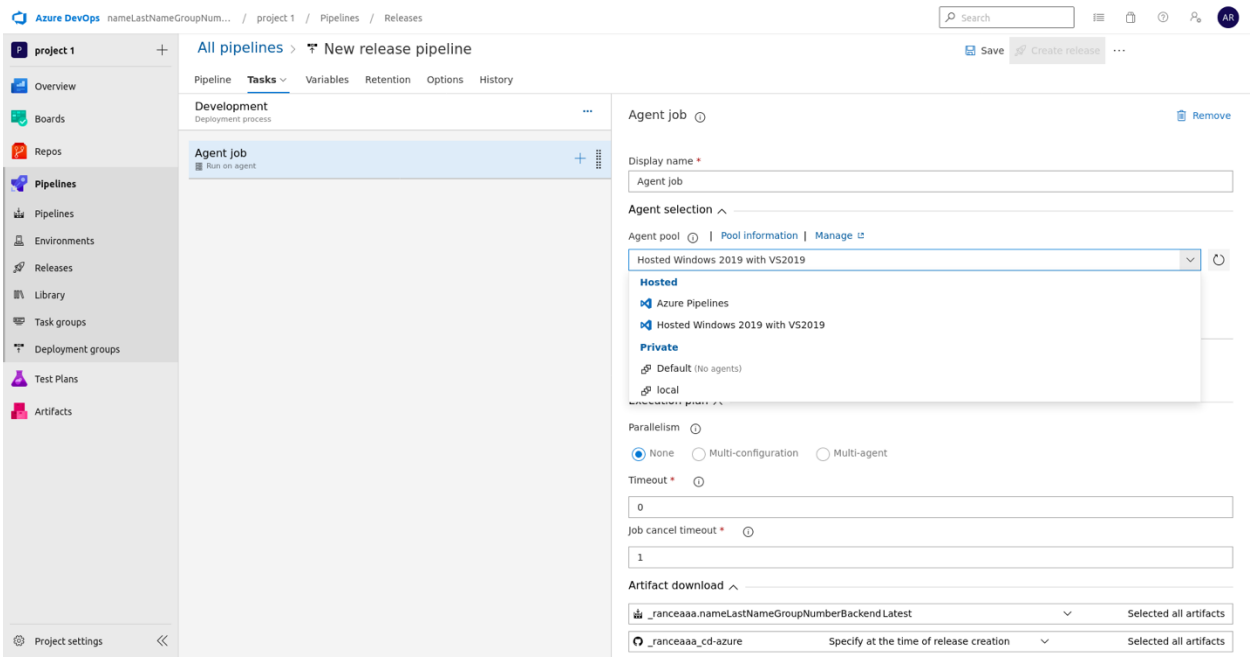

Figure 39. Agent Job Configuration

To add a new task, press the *+ button* next to Agent job, and search for Docker Compose. Add the Docker Compose task, and select you Azure subscription.

The next configuration must to be done, to work with the Docker Compose file we are pulling from our GitHub repository:

1. Change the Docker Compose file path to → $(System.DefaultWorkingDirectory)/_usernameGithub_cd-azure/docker-compose.yml

2. Complete on Action field → Run a Docker Compose command

3. Command → up –d

4. On Advanced Options add the Working Directory → $(System.DefaultWorkingDirectory)/_usernameGithub_cd-azure/

5. Press Save

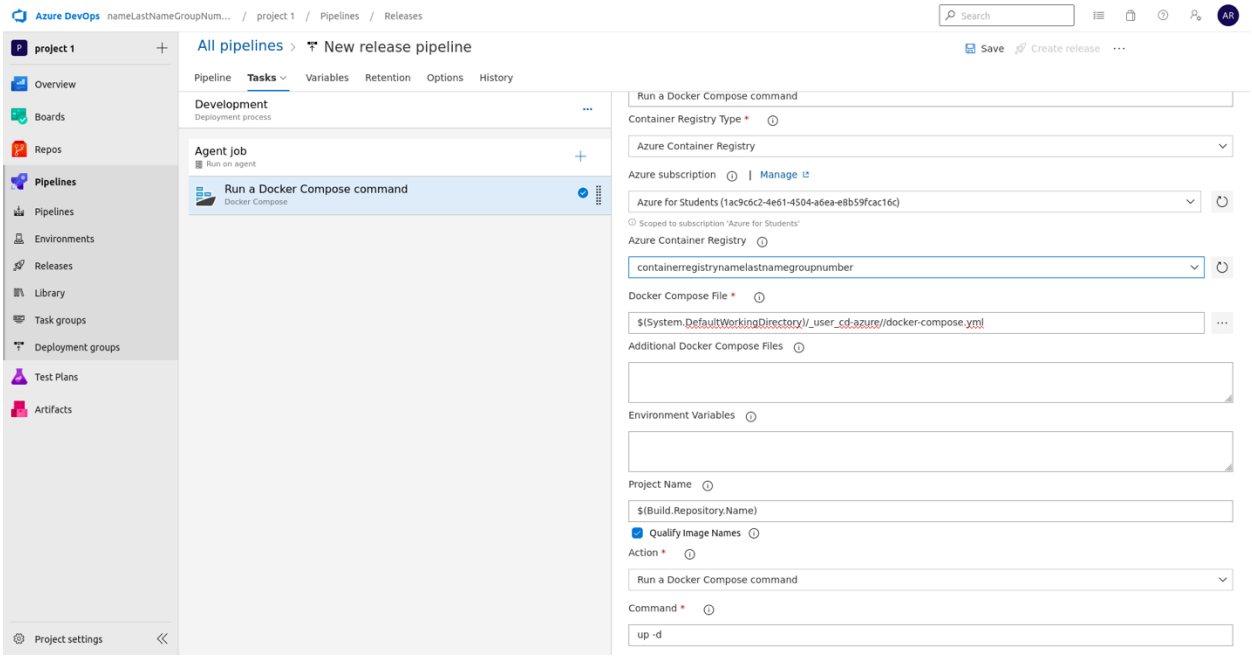

Figure 40. Agent job - Docker Compose Configuration 1

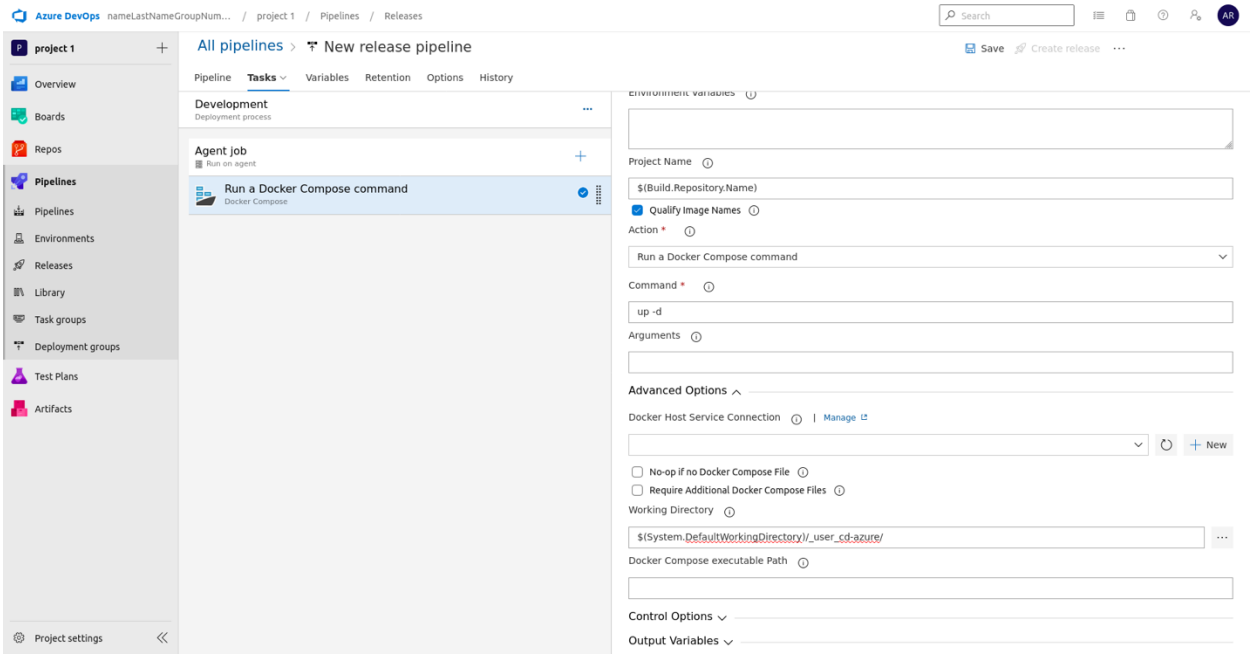

Figure 41. Agent job - Docker Compose Configuration 2

The second task that must be added is Bash. This contains CLI commands responsible for deploying the applications.

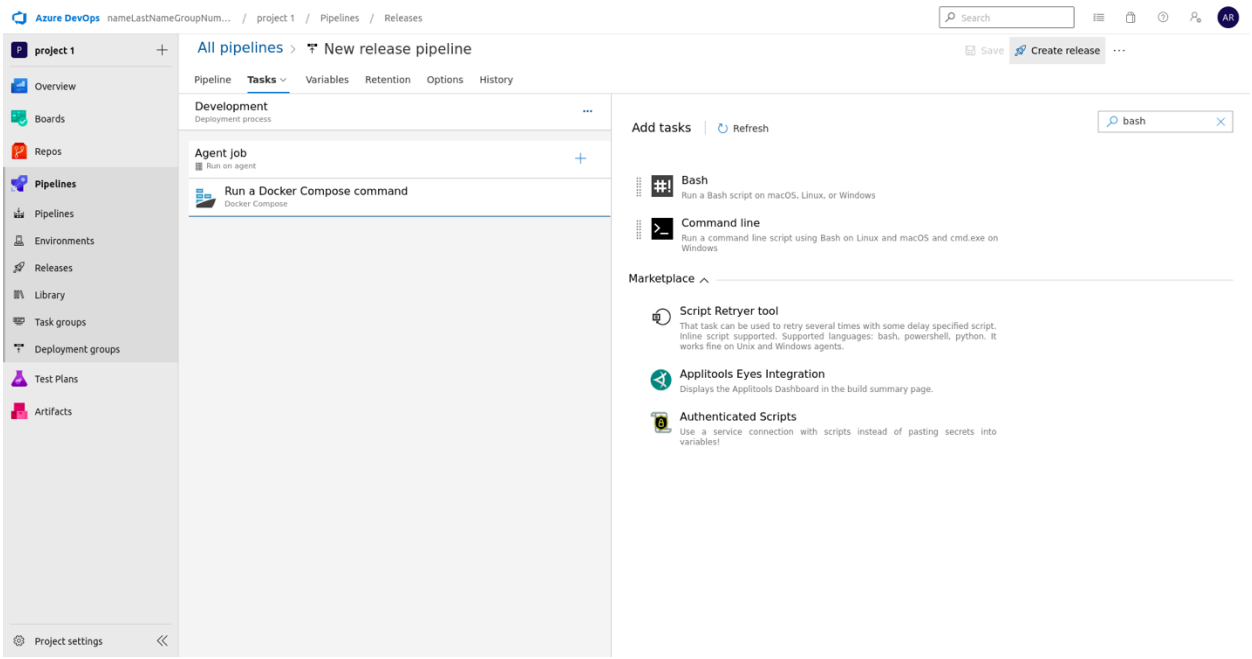

Figure 42. Bash Script

Select the Type to Inline and add the following script:

1. Move to the _usernameGithub_cd-azure folder.

```
cd usernameGithub_cd;
```

2. After the first task is finished (docker compose up), the running containers should be stopped.

```
docker compose down;
docker-compose down;
```

3. Login into the Azure Container Registry:

```
docker login containerregistrynamelastnamegroupnumber.azurecr.io -u
containerregistrynamelastnamegroupnumber -p sTsnMgSQa0dx5LD=qWqEtcIv6zmJSmMP;
```

The credentials can be found in: Home → Resource Groups → nameLastNameGroupNumber → containerregistrynamegroupnumber → Access keys and by enabling the Admin user. Replace the Login server, Username and password with your own details.

4. Create a Docker aci context on your resource group

```
docker context create aci acicontext --resource-group nameLastNameGroupNumber;
```

5. Use the context created above.

```
docker context use acicontext;
```

6. Start the containers

```
docker compose up;
```

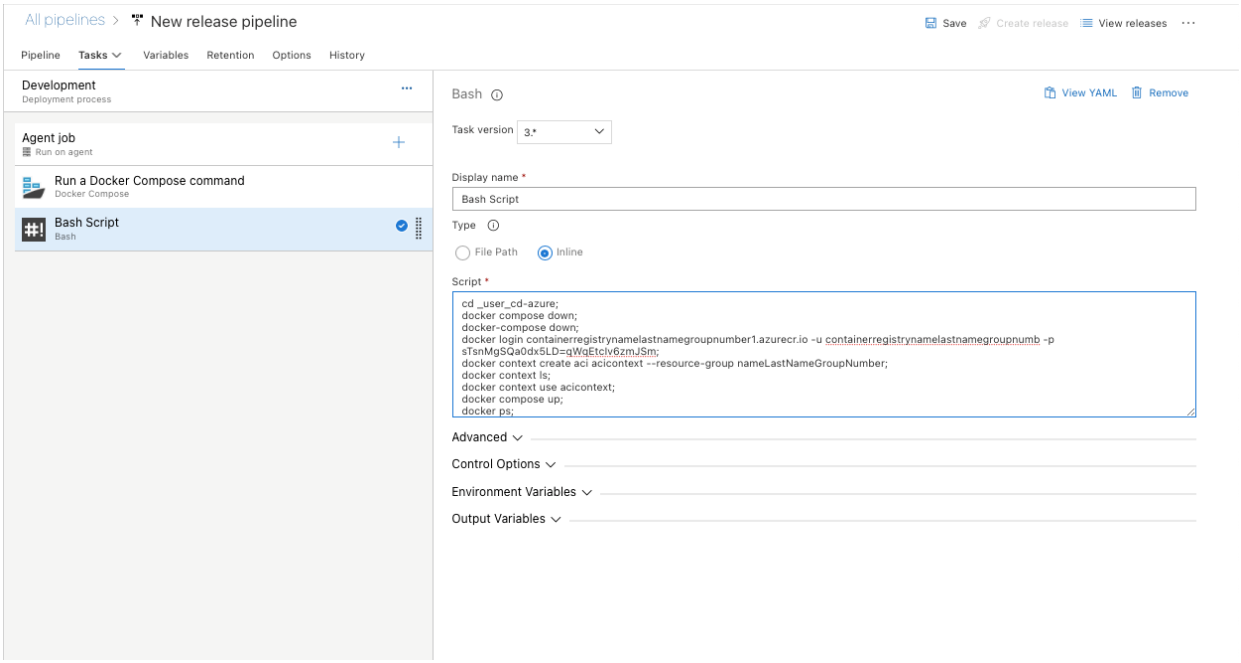

Figure 43. Bash Inline Script

After the Development Release is finished and Saved, press Create release and press Deploy.

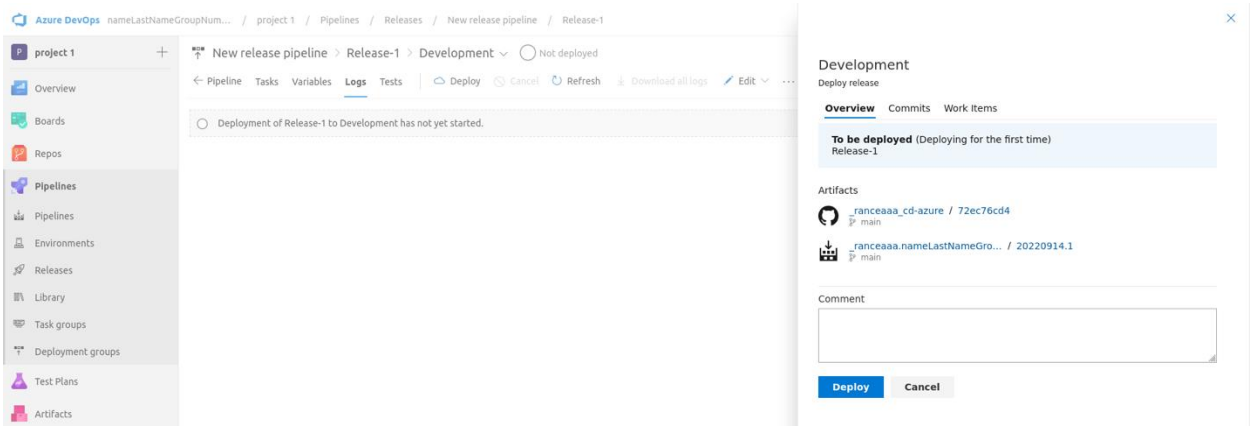

Figure 44. Create Release

When the next push occurs in your GitHub repository, the build pipeline will start, and if it is completed successfully, the CD pipeline starts. The output should be like Figure 45.

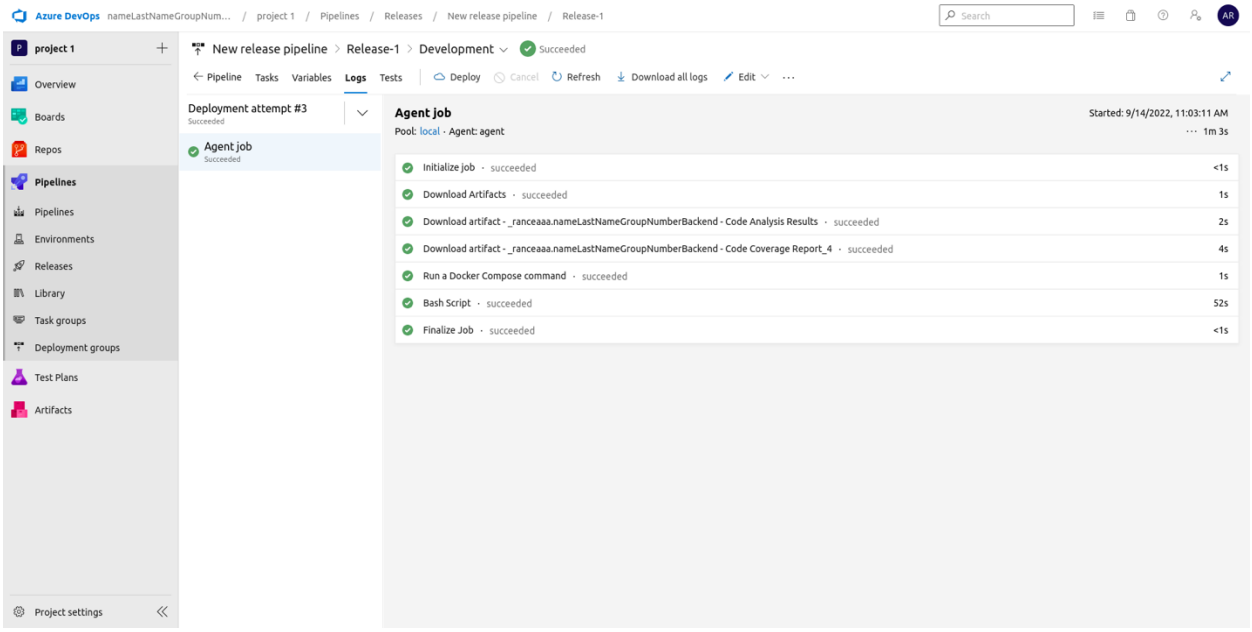

Figure 45. Complete CD Pipeline Logs

If Bash Script log is selected, the IP of the server can be found, and tested by copying it in a browser.

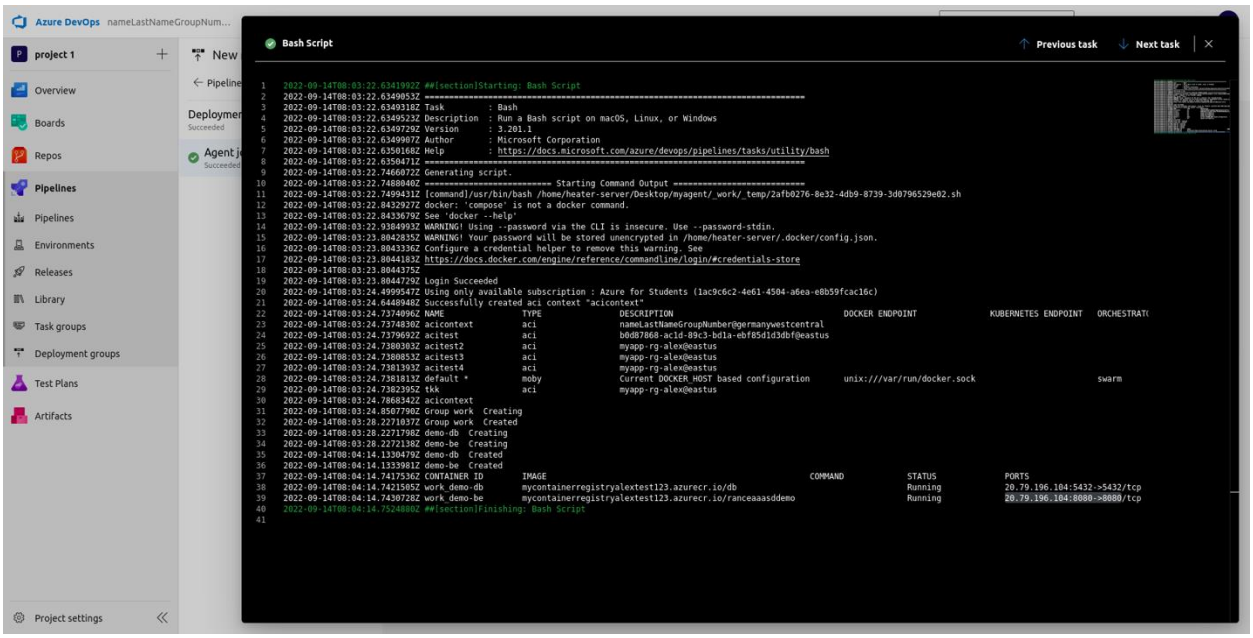

Figure 46. IP of the Deployed Containers

To visualize the deployed application and the containers, go to *Home* → *Resource Groups* → *nameLastNameGroupNumber* → *work* → *Containers.*

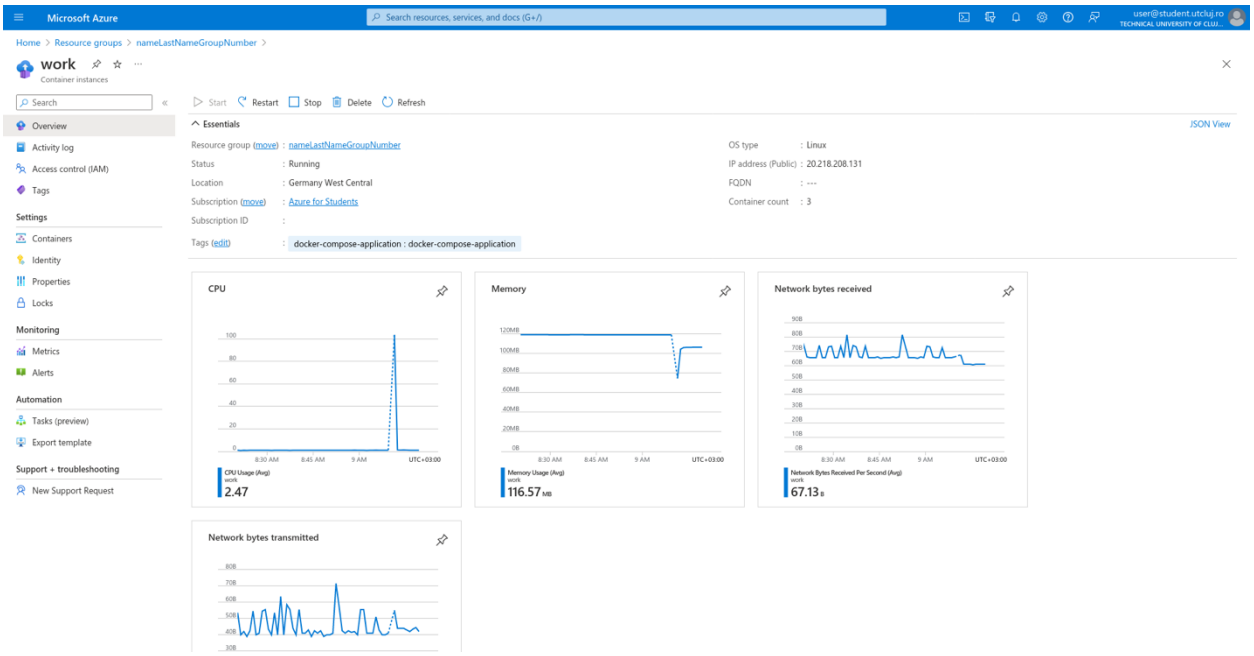

Figure 47. Deployed Application Dashboard

5. Further development

- Apply the configuration from section 4 to deploy the React Application on Azure.
- Modify the host.js file by referencing the public IP address of the Spring App container.
- Deploy the application resulted from Assignment 1 on Azure 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://docs.gitlab.com/ee/user/project/deploy_tokens/
[5] https://developer.okta.com/blog/2020/06/24/heroku-docker-react
[6] https://docs.docker.com/install/linux/docker-ce/ubuntu/
[7] https://www.computerhope.com/jargon/u/user-space.htm
[8] https://en.wikipedia.org/wiki/Hypervisor
[9] https://devopscon.io/blog/docker/docker-vs-virtual-machine-where-are-the-differences/
[10] https://www.docker.com/blog/vm-or-containers/
[11] https://www.sciencedirect.com/topics/computer-science/hypervisors
[12] https://docs.docker.com/registry/
[13] https://docs.docker.com/develop/develop-images/baseimages/
[14] https://en.wikipedia.org/wiki/Virtual_machine