

Candidate number: 10026

# CI/CD Pipeline Portfolio

---

## Table of Contents

- [1. Motivation](#)
- [2. Tools](#)
  - [2.1. Which tools and why](#)
  - [2.2. Pros & Cons](#)
- [3. Pipeline](#)
  - [3.1. Location](#)
  - [3.2. Triggers](#)
  - [3.3. Jobs](#)
    - [3.3.1. Unit testing](#)
    - [3.3.2. Sonar Cloud Analyzing](#)
    - [3.3.3. Building](#)
    - [3.3.4. Deploying](#)
- [4. Iterations](#)
- [5. Experiences](#)
- [6. Further works](#)
- [7. Complete CI/CD Pipeline Configuration File](#)
- [8. Sources & Resources](#)

## 1. Motivation

Today, when working with a software product, you often want a fully functional and automated pipeline that works from code commit to production/test environment. But what are the motivations behind this?

By having a working pipeline you can be more confident that every feature of your software product that is pushed to production is tested and working correctly.

A pipeline makes it much easier to work with continuous integration and continuous delivery (CI/CD) by having everything automated. Everything a developer has to do when he is finished implementing a new feature, is to press a single button that will trigger the pipeline. When the pipeline is complete, the developer and every one else on the team will either get confirmation that everything is still working or that something broke and has to be fixed.

One of the main reasons we want to have CI/CD is to rapidly deliver new features and refactor our code to make it better and more efficient. This will also yield in more rapidly feedback from the end user.

**Side note:** When to push new features to production should often be a business decision. That's why for most pipelines you don't want to automatically deploy it to production. Rather to a staging environment where it can be staged to confirm everything is working and have a working product with all the new features on stand-by ready to be deploy to production when the decision is made.

## 2. Tools

## **2.1. Which tools and why**

In my pipeline I've used five different tools, all working together to create, test, build, deploy and give feedback.

### **2.1.1. Spring boot**

To create a basic web application that can be deployed and tested.

### **2.1.2. Sonar Cloud**

Static code analyzing tool for analyzing the project for security, code smells, test coverage etc... Gives good feedback on the code with a clear dashboard.

### **2.1.3. Docker**

Containerize the application, both the spring boot web app but also the postgresSQL database the spring boot app should be connected to.

### **2.1.4. GitHub Actions**

Setting up a pipeline with triggers based on repository events directly together with the rest of my code base.

### **2.1.5. Azure**

Production environment to host the application. Tested multiple cloud service providers earlier in the semester and ended up liking azure the most. Has a clean dashboard, ease to understand walk-throughs when setting up resources and a lot of documentation.

## **2.2. Pros & Cons**

### **2.2.1. Sonar Cloud**

You might have heard of Sonar Lint for developing programs, giving you feedback in your IDE when your are writing code on code smells, bugs, security etc... Sonar Cloud does the same, only in the cloud. This is very handy and gives you a quick overview on the state of your application as well as feedback on improvements you can make to your code base.

All though it's handy, it is another tool in your pipeline you have to pay for. For small hobby projects you can probably get away without paying, however for bigger projects in companies you have to pay for the service. You have to make the decision if it's worth having.

### **2.2.2. GitHub Actions for GitOps**

One of the nice things with using GitHub Actions is that you have everything in one place. You normally store your source code with version control so why not store your infrastructure and pipeline the same way? However, with this approach you might end up getting locked in to the git provider, making it more difficult to change your/the teams workflow if you want to move from the original git provider.

### 2.2.3. Azure

Many companies today use Azure as their cloud service providers. One of the main reasons is often that they already have other Microsoft services making Azure more similar and ease to understand. It is beneficial to have some experience with Azure, however Azure also makes itself more pleasant to work with due to its extensive documentation.

For some, Azure might seem a bit expensive, but a lot of the costs goes to their Microsoft license which companies sometimes pay for anyways due to having other Microsoft services, making the cost much less expensive.

With Azure you can create your own dashboard, pinning resources to it to get an overview over all the resources you currently have setup. This leads to you, the owner of these resources, being fully aware so you don't have resources running that you do not use.

## 3. Pipeline

The pipeline defines all the steps that should be taken whenever a change is committed to the repository. It also defines when these steps should be triggered.

### 3.1. Pipeline Location

The pipeline configuration files are stored under version control in the [GitHub repository](#) under `.github/workflows/build.yml`

### 3.2. Pipeline Triggers

The pipeline is triggered whenever a change is pushed to the main branch. You can also configure it to be triggered whenever a pull request to the main branch is opened. This is usually how teams work, by creating a new branch where a single developer can work on a single new feature. When it's implemented, the developer opens up a pull request to the main branch, which triggers the pipeline as well as other developers can review the new changes.

Since I've worked on this project alone, I didn't find any reasons to work like this. Instead I just pushed every change directly to the main branch.

```
on:
  push:
    branches:
      - main
```

*Pipeline trigger config*

### 3.3. Pipeline Jobs

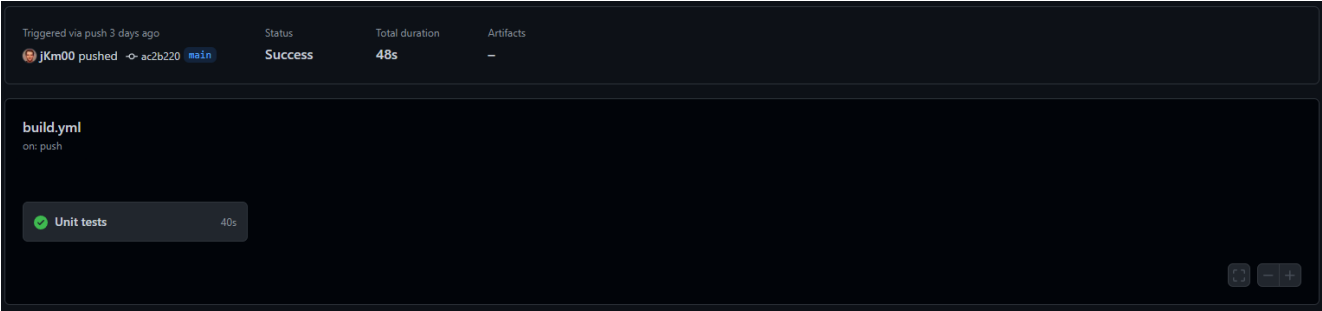
The pipeline consists of four jobs, each responsible for a subtask of the pipeline to be able to run, test, build and deploy the application:

1. [Unit testing](#)

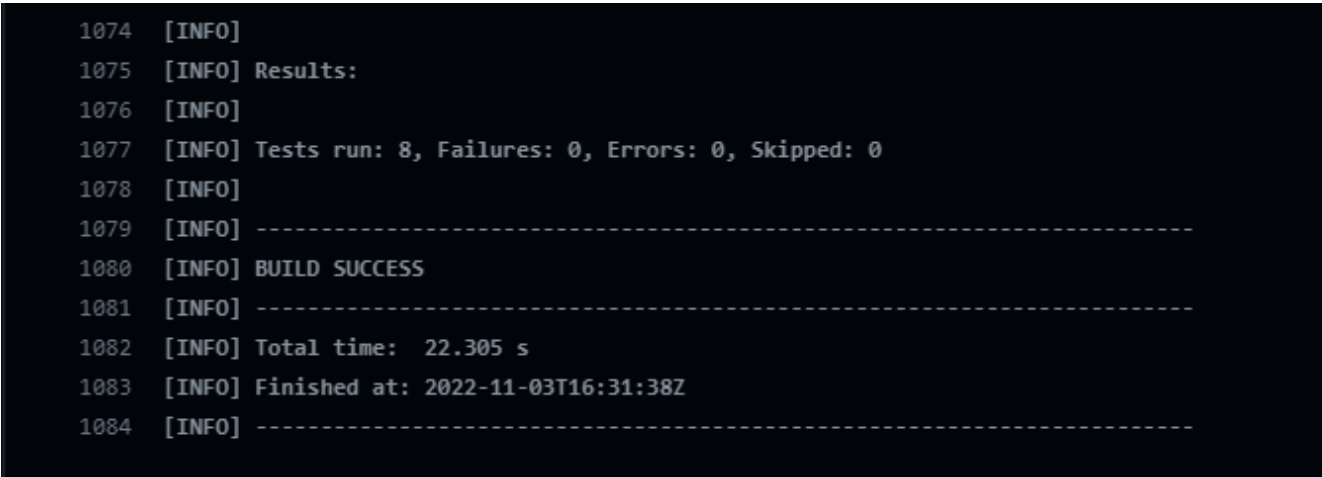
- 2. [Sonar Cloud analyzing](#)
- 3. [Building](#)
- 4. [Deploying](#)

3.3.1. Unit test

The unit test job makes sure all the unit test specified in the test folder of the application is executed and the job is succesfull only if all the unit tests passes. If not the pipeline is canceled with logs of which test failed.

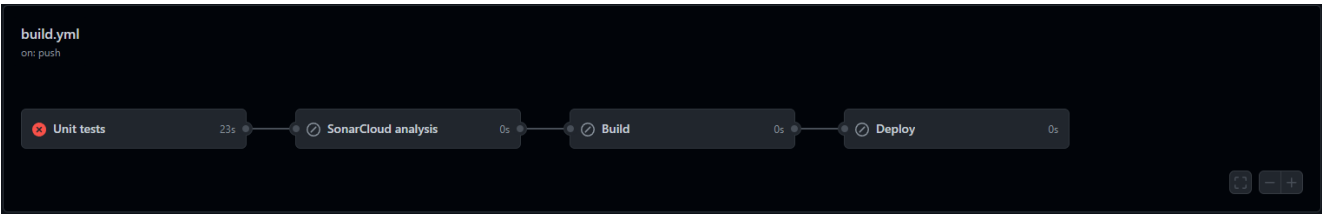


Job: Unit testing, succesfull!



Test results from unit test job

If one or more tests fails, the pipeline is cancelled and the test result is logged in the `test` step of the pipeline. You can navigate your way to it and have a look at what went wrong to make it easier to pin-point what needs to be fixed.



Pipeline canceled when on or more tests fails

```

108 [INFO] Results:
109 [INFO]
110 Error: Failures:
111 Error:   AppControllerTests.testVersionEndpoint:17 expected: <version 2.0> but was: <version 1.0>
112 [INFO]
113 Error: Tests run: 6, Failures: 1, Errors: 0, Skipped: 0
114 [INFO]
115 [INFO] -----
116 [INFO] BUILD FAILURE
117 [INFO] -----
118 [INFO] Total time: 13.773 s
119 [INFO] Finished at: 2022-11-14T13:55:02Z
120 [INFO] -----
121 Error: Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.22.2:test (default-test) on project portfolio-api: There are test failures.
122 Error:
123 Error: Please refer to /home/runner/work/cloud-service-portfolio/cloud-service-portfolio/portfolio-api/target/surefire-reports for the individual test results.
124 Error: Please refer to dump files (if any exist) [date].dump, [date]-jvmRun[N].dump and [date].dumpstream.
125 Error: -> [Help 1]
126 Error:
127 Error: To see the full stack trace of the errors, re-run Maven with the -e switch.
128 Error: Re-run Maven using the -X switch to enable full debug logging.
129 Error:
130 Error: For more information about the errors and possible solutions, please read the following articles:
131 Error: [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/NoIoFailureException
132 Error: Process completed with exit code 1.

```

### Log output when some tests fails

tests:

name: Unit tests

runs-on: ubuntu-latest

steps:

# Check out repo

- uses: actions/checkout@v1

# Set up JDK

- name: Set up JDK

uses: actions/setup-java@v1

with:

java-version: '17'

# Set up maven

- name: Cache Maven packages

uses: actions/cache@v1

with:

path: ~/.m2

key: \${{ runner.os }}-m2-\${{ hashFiles('\*\*/pom.xml') }}

restore-keys: \${{ runner.os }}-m2

# Run tests

- name: Run Tests

run: mvn -B test

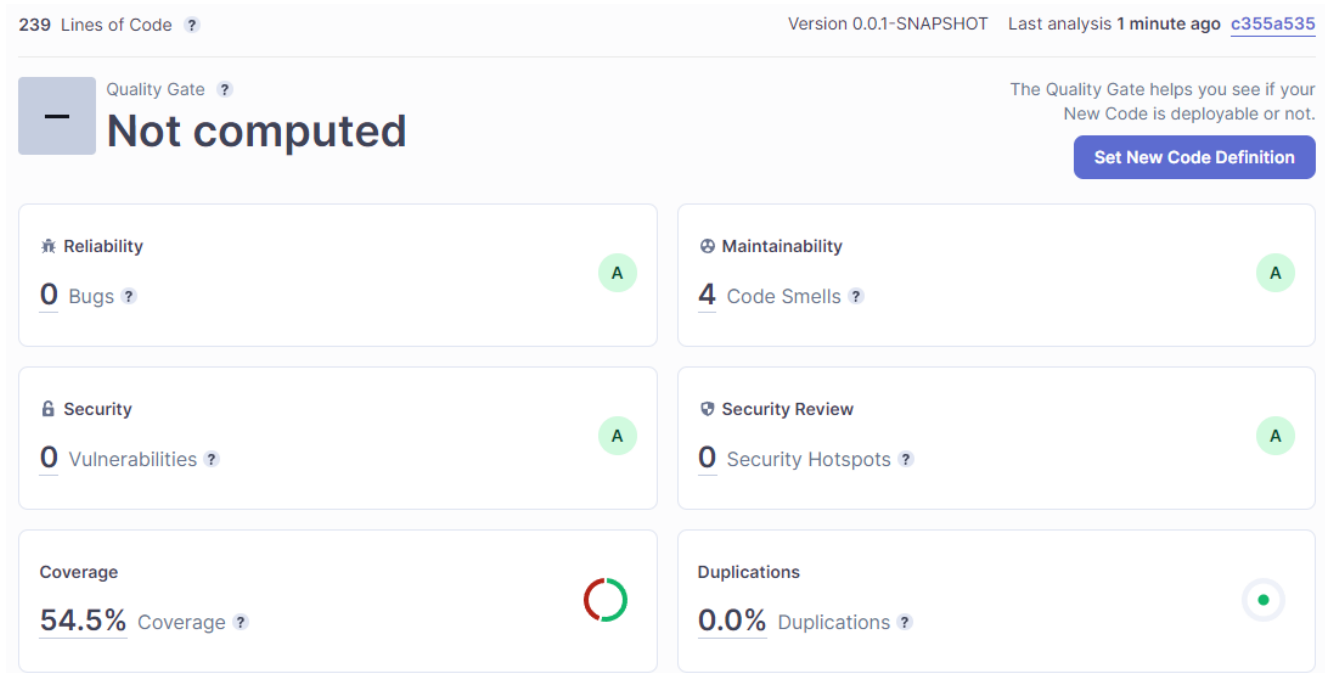
working-directory: ./portfolio-api

### Pipeline config for unit testing

**Note:** Need to specify working directory when executing the tests because the application does not live in the root of the repository.

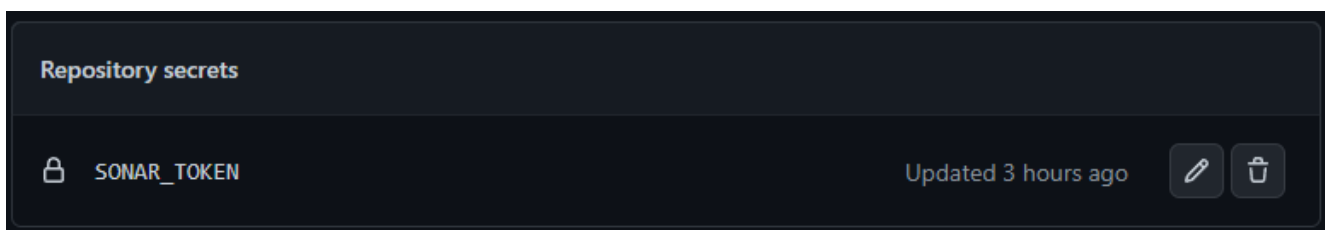
### 3.3.2. Sonar Cloud analyzing

Runs the application with Sonar Cloud to analyze the project and given feedback on bugs, security issues, test coverage, maintainability, etc... A summary of the application state could be found by logging into Sonar Cloud.



### Sonar Cloud Dashboard

For github to get access to Sonar Cloud a token is stored as a repository secret: `settings -> secrets -> actions`. This token is used directly in the pipeline job. Only admins of the repository have access to read, edit or delete this the token.



### Sonar Cloud token stored as repository secret

```
sonar:
  # Make sure test job is succesfull
  needs: tests
  name: SonarCloud analysis
  runs-on: ubuntu-latest
  # Step's required for the job
  steps:
    # Check-out repository
    - uses: actions/checkout@v2
      with:
        fetch-depth: 0 # Shallow clones should be disabled for a better relevancy
of analysis
    # Sets up JDK
    - name: Set up JDK
      uses: actions/setup-java@v1
      with:
        java-version: '17'
    # Sets up SonarCloud cache
```

```



- name: Cache SonarCloud packages
  uses: actions/cache@v1
  with:
    path: ~/.sonar/cache
    key: ${{ runner.os }}-sonar
    restore-keys: ${{ runner.os }}-sonar
# Sets up Maven cache
- name: Cache Maven packages
  uses: actions/cache@v1
  with:
    path: ~/.m2
    key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
    restore-keys: ${{ runner.os }}-m2
# Uses SonarCloud to analyze the project
- name: Build and analyze
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # Needed to get PR information,
    if any
    SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
  run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -
Dsonar.projectKey=jKm00_cloud-service-portfolio -Dspring.profiles.active=test
  working-directory: ./portfolio-api

```

*Pipeline config for Sonar Cloud analyze*

### 3.3.3. Building

Before the app can be deployed, it needs to be built. That's what this job does. After the job has build the application, the **.jar** file is uploaded as an artifact, with name **api**, so it can be shared across the pipeline jobs.

| Artifacts  |         |   |
|--|---------|---|
| Produced during runtime  |         |   |
| Name   | Size    |   |
|  <b>api</b> | 38.4 MB |  |

**Uploaded artifact by pipeline job** *(can be downloaded and executed)*

```

build:
  # Make sure sonar job is succesfull
  needs: sonar
  name: Build
  runs-on: ubuntu-latest
  steps:
    #Check-out repository
    - uses: actions/checkout@v2
    # Set up JDK
    - name: Set up JDK
      uses: actions/setup-java@v3

```

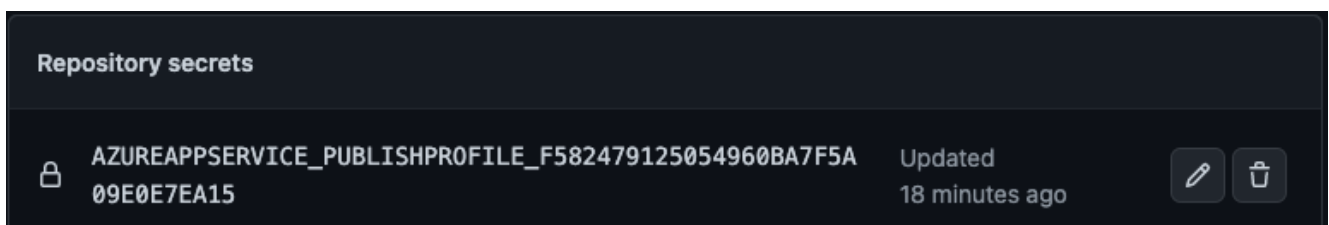
```
with:
  distribution: 'temurin'
  java-version: '17'
  cache: 'maven'
#Build the application using Maven
- name: Build with Maven
  run: mvn -B package -DskipTests --file pom.xml
  working-directory: ./spring-boot
# Upload build version of application
- name: Upload JAR
  # This uploads artifacts from your workflow allowing you to share data
  # between jobs and store data once a workflow is complete.
  uses: actions/upload-artifact@v2
  with:
    name: api
    # From this path
    path: spring-boot/target/skytjenester-docker-demo-1.0.jar
```

*Pipeline config for building*

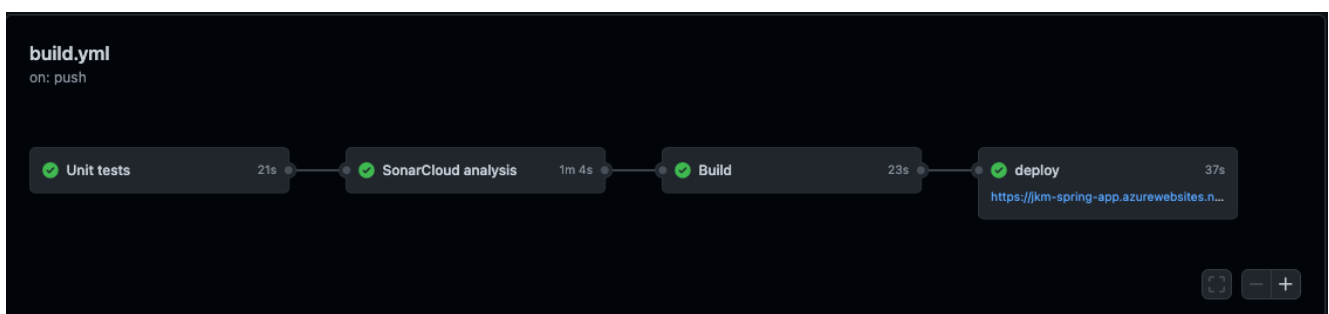
### 3.3.4. Deploying

The job responsible for deploying the spring boot application from the repository to the azure web app. Uses the artifact generated from the previous job (build) and uploads it to the azure web app before it's ran.

For github to get access to the azuer web application another secret has to be stored in the repository:

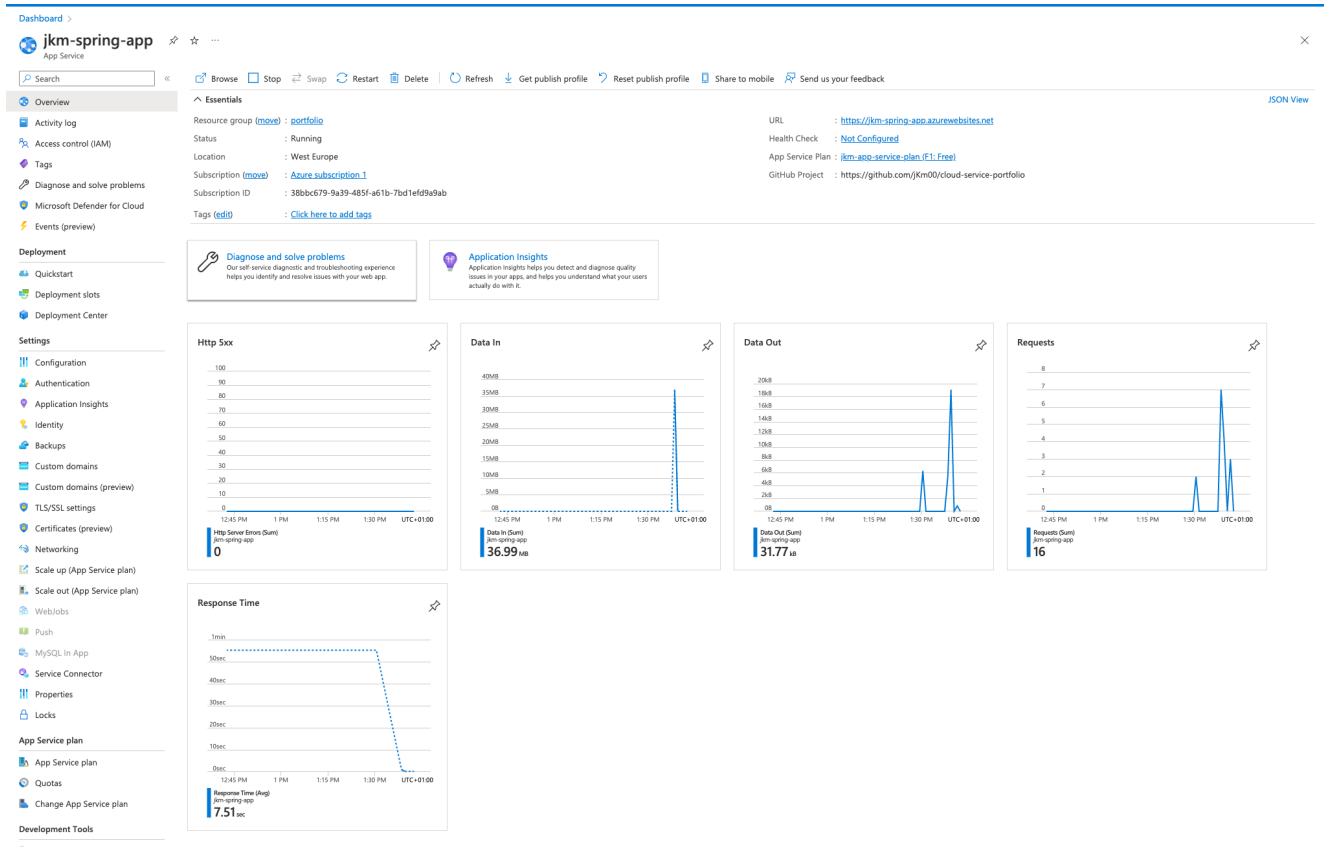


### Azure secret stored in repository

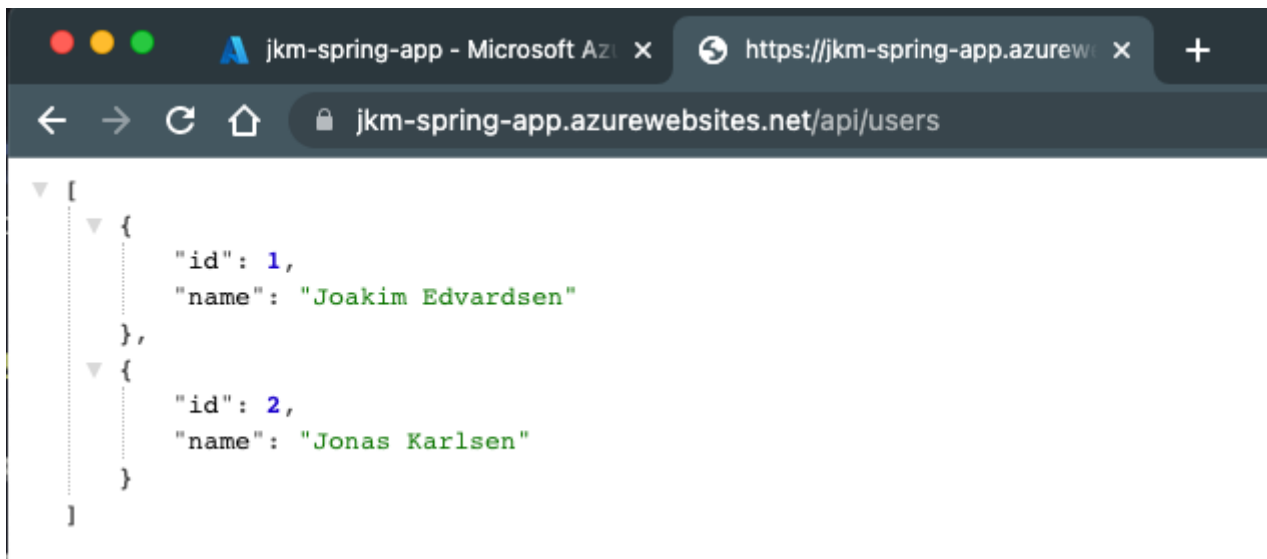


### Succesfull Pipeline with Azure Deployment





## Azure Web App Overview



## Accessing api endpoint from azure web app

```
deploy:
  needs: build
  name: Deploy
  runs-on: ubuntu-latest
  environment:
    name: 'Production'
    url: ${ steps.deploy-to-webapp.outputs.webapp-url }

steps:
  - name: Download artifact from build job
```

```

    uses: actions/download-artifact@v2
    with:
      name: api

  - name: Deploy to Azure Web App
    id: deploy-to-webapp
    uses: azure/webapps-deploy@v2
    with:
      app-name: 'jkm-spring-app'
      slot-name: 'Production'
      publish-profile: ${{
secrets.AZUREAPPSERVICE_PUBLISHPROFILE_F582479125054960BA7F5A09E0E7EA15 }}
      package: '*.jar'

```

*Pipeline config for deployment*

## 4. Iterations

1. First I created the basic spring boot applicaiton with some endpoints. I wanted to connect the application to a database to challenge myself to have more than one service. In the beginning I created the app with an in memory database with the thought of moving this to a postgresQL later in production. I would anyways need the in memory database configuration for testing to make sure I would have consistant test results.
2. Next I started creating the pipeline. The first thing I wanted to complete was automated testing. For this I also needed to implement some unit test that would be run in the pipeline. The pipeline configuration was pretty straight forward. Just needed to setup a VM where I could execute all the tests.
3. After doing some research, I found an iteresting tool, Sonar Cloud, that we were introduced to in the first year, but a tool that I had forgot about. After rediscovering this I wanted to implement it in the pipeline to get automated analyzing of the applicaiton. There were some challenges with this step like getting a token and storing it as a secret as well as actually using that token in the pipeline itself.
4. The next and final step I wanted to complete was deploying the application, however I needed to build the application first. So this became a naturall step by itself. The main challenge with this step were to figure out how to store the built applicaiton so I could use it later to deploy.
5. Now I could takle the finally step, deploying. I had tested some cloud providers earlier in the semester and found that I liked azure the most. That's why I ended up using azure for this portfolio as well. Before I could configure the pipeline to automatically deploy, I had to configure a web application in azure that I could deploy my app to. This was done using azures portal interface. When this was created, azure automatically genereted a workflow file that I could just merge into my own.

*Everything was done with version controll, meaning you can go to the [GitHub repository](#) and view all the commits along the way as well as a history of all the executed workflows under the actions tab.*

## 5. Experiences

I found it pretty easy and intuitivt to work with GitHub actions and getting it set up with triggers so the pipeline would execute whenever a change was made to the code base. For me, it just makes sence to use the

same tools for your CI/CD configuration as you use for your application and services, all though I also see that there can be some disadvantages with this approach.






I had to do some troubleshooting along the way, especially when configuring pipeline jobs that had to be integrated with third parties like Sonar Cloud and Azuer, but both GitHub and Azuer has a lot of documentation that help along the way. The main difficulty was understanding how repository secrets worked and how I could use them in the pipeline.

I also tried to implement some docker functionality at the end. I made the `Dockerfile` for the spring boot application and the `docker-compose` file that spins up a postgresSQL database and the spring boot. Everything works locally, but I had some trouble setting it up with Azure. This would have been something I would have had to look more into to be able to achieve.

## 6. Further Works

As mentioned I started containerizing the application and got everything running locally. With more time working with this pipeline, I would have used the containerized versions of both the spring boot application and a postgresSQL database and ran it using the ``docker-compose.yml`` on azure, instead of just executing the ``jar`` file created from the build job of the pipeline (which runs an in memory database).

Here are the configuration files for the containerization:

|                          |   |                          |               |      |  |  |  |  |
|--------------------------|---|--------------------------|---------------|------|--|--|--|--|
| <input type="checkbox"/> | <div><div></div><div>spring-boot</div><div>2 containers</div></div>  | -                        | Running (2/2) | -    | <div><div></div><div></div><div></div></div> | <div><div></div><div></div><div></div></div> | <div><div></div><div></div><div></div></div> |  |
| <input type="checkbox"/> | <div><div></div><div>database</div><div>d88d34ab622f </div></div> | postgres:12.2            | Running       | 5432 | 34 seconds ag                                | <div><div></div><div></div><div></div></div> | <div><div></div><div></div><div></div></div> | <div><div></div><div></div><div></div></div> |
| <input type="checkbox"/> | <div><div></div><div>api</div><div>0fb3496b05c5 </div></div>      | jkm00/spring-boot:latest | Running       | 80   | 22 seconds ag                                | <div><div></div><div></div><div></div></div> | <div><div></div><div></div><div></div></div> | <div><div></div><div></div><div></div></div> |

### Docker compose running locally

```
# Dockerfile for the backend in the production environment
FROM amazoncorretto:19.0.1-alpine

WORKDIR /app

# Copy the jar file to the container
COPY target/*.jar app.jar

# Run the jar with the production application properties
ENTRYPOINT ["java", "-Dspring.profiles.active=${SPRING_PROFILE_ACTIVE}", "-jar", "app.jar"]
```

### Dockerfile configuration

```
version: '3.5'
services:
  api:
    container_name: api
```

```

image: jkm00/spring-boot
ports:
  - 80:8080
environment:
  - 'SPRING_PROFILE_ACTIVE=prod'
  - POSTGRES_PORT=${POSTGRES_PORT}
  - POSTGRES_DB=${POSTGRES_DB}
  - POSTGRES_USER=${POSTGRES_USER}
  - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
# Make sure database is up and running before starting the api
depends_on:
  db:
    condition: service_healthy

db:
  container_name: database
  image: postgres:12.2
  restart: always
  environment:
    - POSTGRES_USER=${POSTGRES_USER}
    - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    - POSTGRES_DB=${POSTGRES_DB}
  ports:
    - ${POSTGRES_PORT}:5432
  healthcheck:
    test: ['CMD-SHELL', 'pg_isready']
    interval: 10s
    timeout: 5s
    retries: 5

```

*Docker compose config file*

## 7. Complete CI/CD Pipeline Configuration File

```

name: CI/CD Pipeline
on:
  push:
    branches:
      - main

jobs:
  # Test the application
  tests:
    name: Unit tests
    runs-on: ubuntu-latest
    steps:
      # Check out repo
      - uses: actions/checkout@v2
      # Set up JDK
      - name: Set up JDK
        uses: actions/setup-java@v3

```

```
    with:
      distribution: 'temurin'
      java-version: '17'
      cache: 'maven'
  # Run tests
  - name: Run Tests
    run: mvn -B test
    working-directory: ./spring-boot

# Sona's job
sonar:
  # Make sure test job is succesfull
  needs: tests
  name: SonarCloud analysis
  runs-on: ubuntu-latest
  # Step's required for the job
  steps:
    # Check-out repository
    - uses: actions/checkout@v2
      with:
        fetch-depth: 0 # Shallow clones should be disabled for a better
relevancy of analysis
    # Set up JDK
    - name: Set up JDK
      uses: actions/setup-java@v3
      with:
        distribution: 'temurin'
        java-version: '17'
        cache: 'maven'
    # Sets up SonarCloud cache
    - name: Cache SonarCloud packages
      uses: actions/cache@v1
      with:
        path: ~/.sonar/cache
        key: ${{ runner.os }}-sonar
        restore-keys: ${{ runner.os }}-sonar
    # Set up maven
    - name: Cache Maven packages
      uses: actions/cache@v1
      with:
        path: ~/.m2
        key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
        restore-keys: ${{ runner.os }}-m2
    # Uses SonarCloud to analyze the project
    - name: Build and analyze
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # Needed to get PR
information, if any
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
      run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
-Dsonar.projectKey=jKm00_cloud-service-portfolio
      working-directory: ./spring-boot

# Build the application for production
```

```
build:
  # Make sure sonar job is succesfull
  needs: sonar
  name: Build
  runs-on: ubuntu-latest
  steps:
    #Check-out repository
    - uses: actions/checkout@v2
    # Set up JDK
    - name: Set up JDK
      uses: actions/setup-java@v3
      with:
        distribution: 'temurin'
        java-version: '17'
        cache: 'maven'
    #Build the application using Maven
    - name: Build with Maven
      run: mvn -B package -DskipTests --file pom.xml
      working-directory: ./spring-boot
    # Upload build version of application
    - name: Upload JAR
      # This uploads artifacts from your workflow allowing you to share data
      # between jobs and store data once a workflow is complete.
      uses: actions/upload-artifact@v2
      with:
        name: api
        # From this path
        path: spring-boot/target/skytjenester-docker-demo-1.0.jar

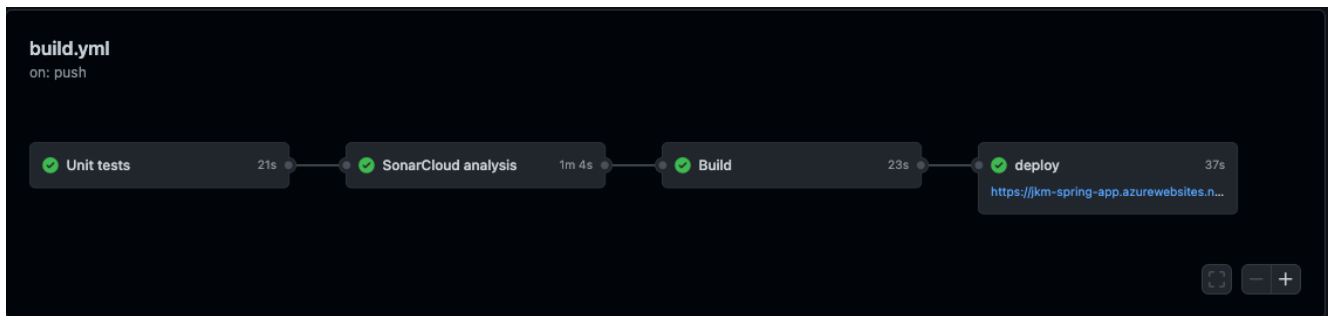
# Deploy application to azure
deploy:
  needs: build
  name: Deploy
  runs-on: ubuntu-latest
  environment:
    name: 'Production'
    url: ${ steps.deploy-to-webapp.outputs.webapp-url }

  steps:
    - name: Download artifact from build job
      uses: actions/download-artifact@v2
      with:
        name: api

    - name: Deploy to Azure Web App
      id: deploy-to-webapp
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'jkm-spring-app'
        slot-name: 'Production'
        publish-profile: ${ secrets.AZUREAPPSERVICE_PUBLISHPROFILE_F582479125054960BA7F5A09E0E7EA15 }
        package: '*.jar'
```

---

### Config file



---

### Complete Pipeline Summary

## 8. Sources & Resources

- [Pipeline configuration guide](#)
- [GitHub Actions: Java with maven guide](#)
- [Sonar Cloud Github Actions](#)
- [Azure / GitHub Pipeline](#)
- [My GitHub Repository](#)