

Non-Functional requirements

[Performance and scalability.](#)

[Portability and compatibility.](#)

[Reliability](#)

[Maintainability, availability.](#)

[Security.](#)

[Usability.](#)

Performance and scalability.

Using Quarkus for java applications to improve performance. Because it implements eclipse microprofile it is ideal for creating microservices. Quarkus makes the application more efficient, with quick start up times, faster development as it has live reload, it's easier to learn and use, and consumes less machine resources (memory) making the API easily scalable when deployed.

Quarkus uses Hibernate with Panache as an ORM framework to write the sql queries faster from the entities.

Quarkus detects in our configuration that we have a database and it starts the DB for us.

Portability and compatibility.

Using Java so it can be run anywhere thanks to the JVM.

Also using Quarkus as it is built to run in containers, so if the application is run in the cloud this is a good option. It makes it easier to deploy to openshift by dockerising the application.

Reliability

Quarkus implements eclipse.microprofile which comes with metrics that can be used through annotations such as `@Counted` and `@Timed`, allowing counting the invocations of the annotated object and counting of

number of calls respectively. I didn't use them, but this should be added in the next iteration as it will allow the creation of dashboards in Grafana labs and understanding of how the system performs. This feedback from the data analysis will in turn be used to improve the service.

API includes testing to the endpoints - integration testing.

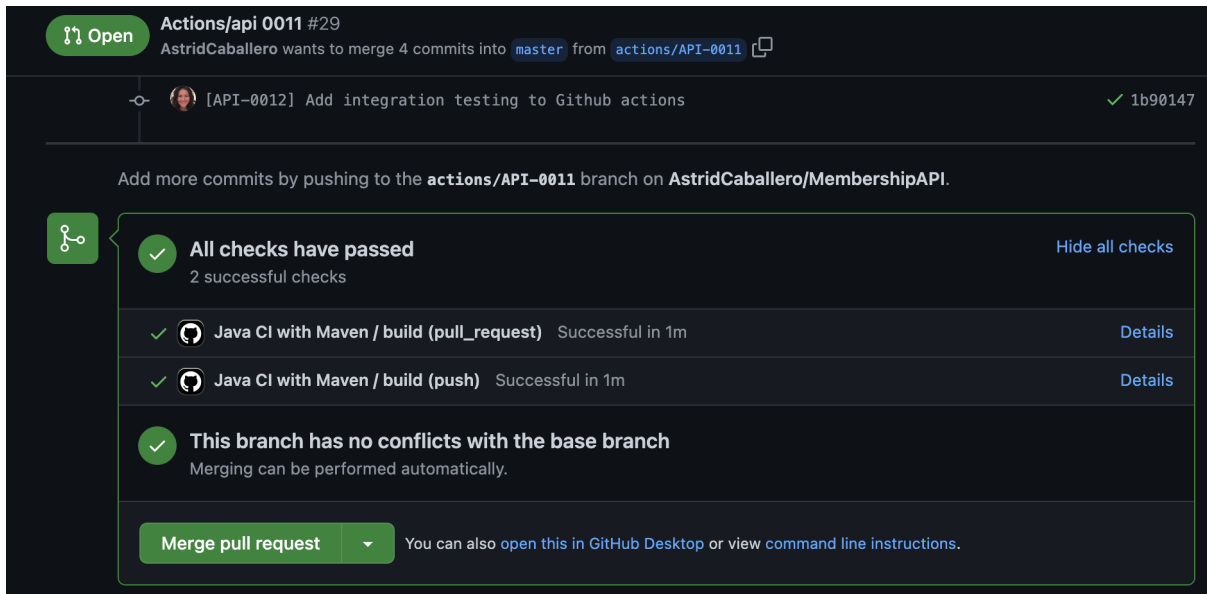
Next iteration needs to:

- Add unit test and surefire dependency.
- Add test coverage with Jacoco.

Maintainability, availability.

Creating a CI/CD pipeline for example github actions or Jenkins and Openshift, where the quality of the code can be tested before merging with the main branch. Openshift will deploy a stable version of the service and it will take care of its availability and downtime by providing the number of pods that are needed according to traffic.

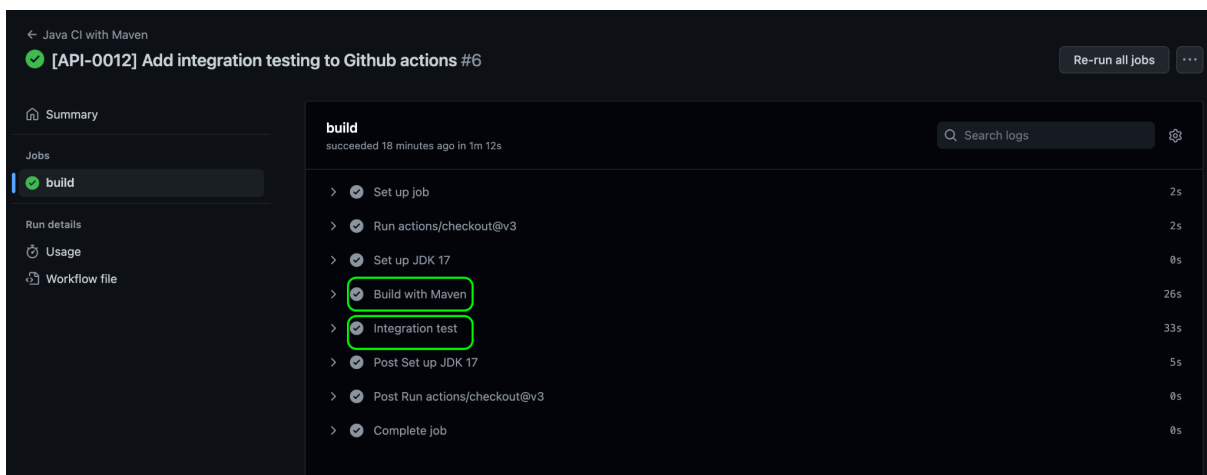
This prototype includes a simple CI pipeline using githubs, each time a change is push to the repository in github it will check the build and the tests:



In the image above, the checks are passing:

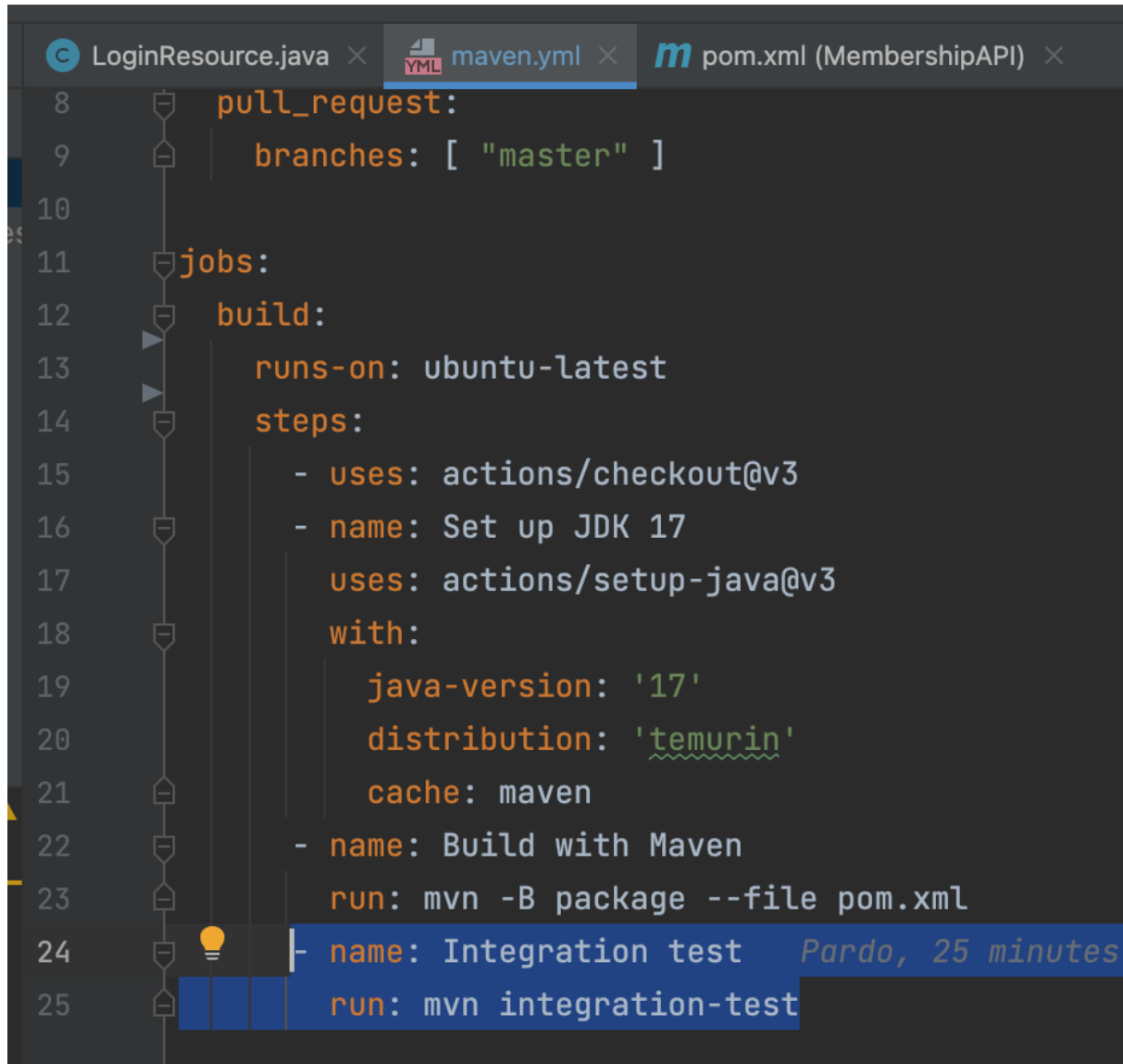
- The build for when the branch was pushed
- The build for the Pull Request - PR

If we go to details of the PR:



we can see that both the build and test have passed.

In the next iteration the .yml file needs to be edited so the test check is not inside the build job (see below, line 24) and it is a job in itself in order to be visible directly in the PR:



```
8 pull_request:
9   branches: [ "master" ]
10
11 jobs:
12   build:
13     runs-on: ubuntu-latest
14     steps:
15       - uses: actions/checkout@v3
16       - name: Set up JDK 17
17         uses: actions/setup-java@v3
18         with:
19           java-version: '17'
20           distribution: 'temurin'
21           cache: maven
22       - name: Build with Maven
23         run: mvn -B package --file pom.xml
24       - name: Integration test Pardo, 25 minutes
25         run: mvn integration-test
```

Security.

Next iteration needs to :

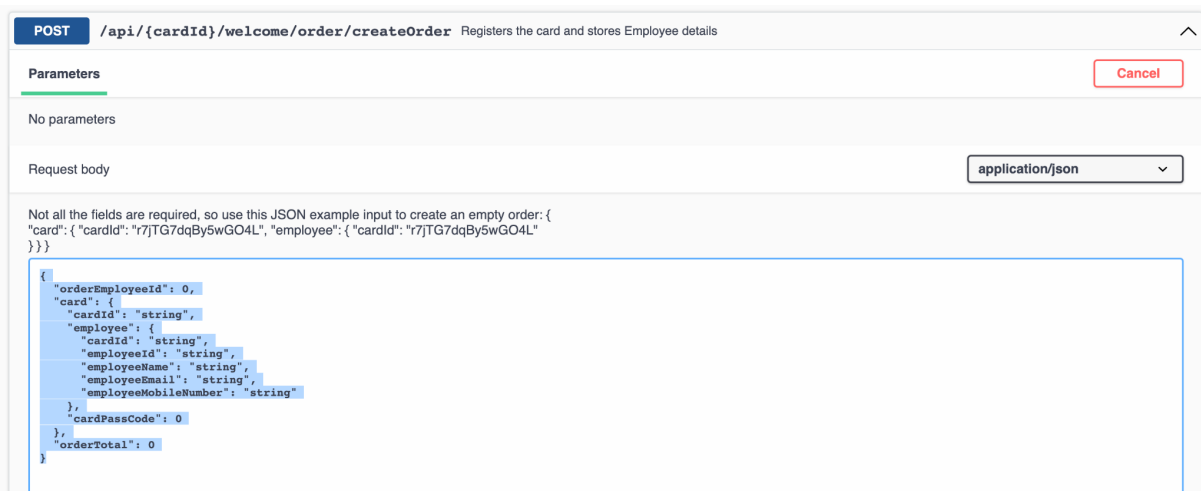
- Create roles in the API, to allow access to the right people.
- Encrypt the passwords, e.i adding salt and pepper.

- The API also needs to implement authentication.
- Make data validation robust.

Usability.

For the prototype scope, I used Swagger for the user to interact with the API easily. In the Swagger UI the endpoints are displayed along with a description and its details

Each request provides input examples to make it easy for the user to test the API:



The image shows a Swagger UI interface for a POST endpoint. The endpoint path is `/api/{cardId}/welcome/order/createOrder` with a description "Registers the card and stores Employee details". The "Parameters" section is empty. The "Request body" section is set to "application/json" and contains a note: "Not all the fields are required, so use this JSON example input to create an empty order: { 'card': { 'cardId': 'r7jTG7dqBy5wGO4L', 'employee': { 'cardId': 'r7jTG7dqBy5wGO4L' } } }". Below the note is a large text area with a JSON schema template highlighted in blue.

```
{
  "orderEmployeeId": 0,
  "card": {
    "cardId": "string",
    "employee": {
      "cardId": "string",
      "employeeId": "string",
      "employeeName": "string",
      "employeeEmail": "string",
      "employeeMobileNumber": "string"
    },
    "cardPassCode": 0
  },
  "orderTotal": 0
}
```

In the image above we can see the POST request along with a description. It says that the request is a body in JSON format and it provides a template (highlighted in blue) of the entity schema to be edited by the user. It also has a note above the highlighted JSON giving an input example that can be used, so the user can copy it and replace the highlighted JSON with the input example for ease.

Each request also documents the responses:

Responses		
Code	Description	Links
201	<div>Order created</div> <div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value</div> <div><pre>{ "orderEmployeeId": 1, "card": { "cardId": "r7jTG7dqBy5wG04L", "employee": { "cardId": "r7jTG7dqBy5wG04L" }, "cardPassCode": 0 }, "orderTotal": 0 }</pre></div>	No links
500	Internal Server Error	No links