deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Pedro Ramos [107348], João Luís [107403]*, Daniel *Madureira [107603], Rodrigo Aguiar [108969]
v2024-05-16

1 Project management

1.1 Team and roles

Role	Name	Email	NMEC
Team Manager	Rodrigo Aguiar	rodrigoaguiar96@ua.pt	108969
Product Owner	João Luís	jnluis@ua.pt	107403
QA Engineer	Daniel Madureira	daniel.madureira@ua.pt	107603
DevOps Master	Pedro Ramos	p.ramos@ua.pt	107348

1.2 Agile backlog management and work assignment

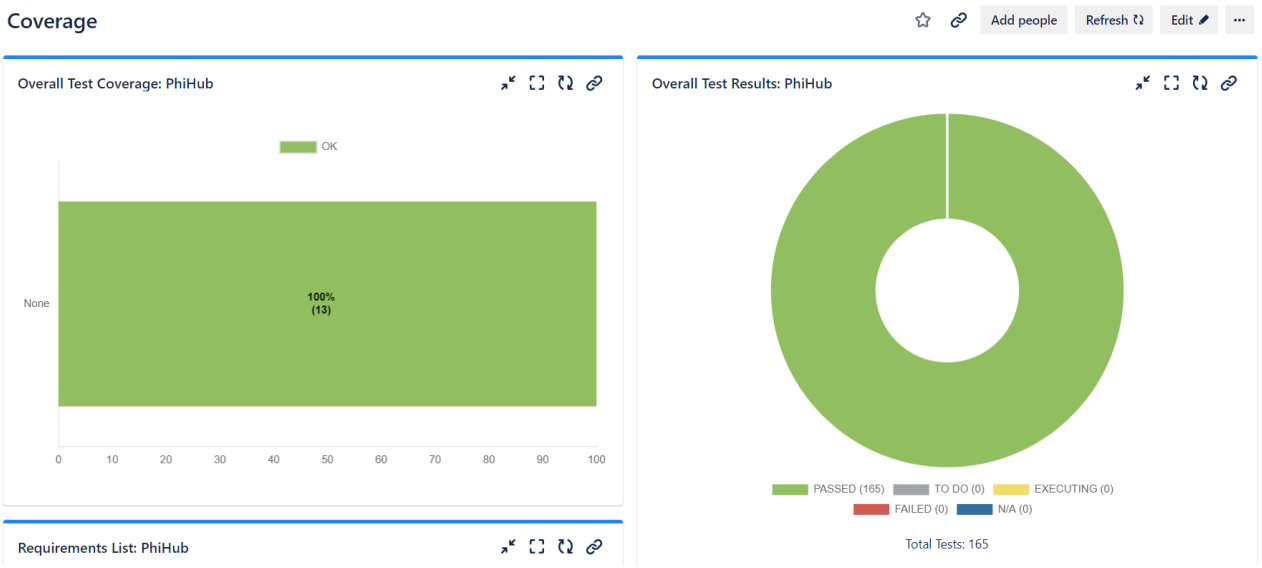
We work on weekly sprints and we do all backlog management on Jira.

At the start of each sprint, the team gathers and decides what user stories will be worked on during this sprint.
The team manager then distributes tasks for each of the team members to work on the sprint, making sure that they are distributed fairly.

When we start the development of a user story, we first create various subtasks assigned to it.

These subtasks are directly related to the development and testing effort of the user story, being for example the development of the backend code needed to support this feature or the functional tests of the frontend component.

We also integrated Xray tests in JIRA which allows us to have the automatic test results integrated within Jira and linked to the given user stories.



Out of the 14 user stories planned initially, all of them were completed and covered with tests, having a total of 165 tests.

2 Code quality management

2.1 Guidelines for contributors (coding style)

All contributions to our code repository should follow these rules:

- All the class names must be in pascal case (HelloWorld.java, for example);
- Package names must be declared at the start of the file, without upper case or numbers;
- Use specific imports, without * ;
- Use of camel case in variable declaration (someVariable, for example)
- Proper code formatting and linting, with the use of the Prettier code formatting tool;
- Limit the scope of variables to the least needed
- The use of lombok to auto generate constructors, getters, setters and toString functions
- The code should be properly commented so it's easy to understand
- All code contributions should be supplied with automated tests using the JUnit framework.
- Use of the Sonarlint VSCode extension is highly recommended to prevent code smells or security issues

2.2 Code quality metrics and dashboards

For the *static code analysis*, we use Sonar Cloud with the Quality Gate "Sonar Way", which is the default QG defined. The dashboard can be consulted on this link.

We didn’t feel the need to define a personalized Quality Gate as this one perfectly fits our needs and assures good quality in the developed code.

This QG has the following conditions, that need to be respected in order to pass the QG:

- No new bugs are introduced;
- No new vulnerabilities are introduced;
- New code has limited technical debt;
- All new security hotspots are reviewed;
- Coverage is greater than or equal to 80.0%;
- Duplicated Lines (%) is less than or equal to 3.0%.

The analysis is done every time a pull request is made or a push to the main branch and the results are automatically published to sonarcloud.

For the frontend components, automatic analysis by sonarcloud is also done to ensure that the developed code is of good quality, presenting few code smells.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

GitHub flow is the workflow adopted for development.

For each task created on the Jira platform, a branch is created to develop that task, and when it's done, it's created a Pull Request to the main branch of that submodule. The pull request must then be reviewed by another member of the team, to ensure that it meets the necessary quality standards and pass the automatic code analysis done by Sonarcloud in order to be merged into the main branch.

For a user story to be considered done, the team complies to the following criteria:

- The developed solution meets **all** the requirements for the user story;

- It is supplied with automatic tests, both unit and integration tests;

- If its a frontend functionality, functional tests must also be provided;

- Test coverage must be above 80%;

- The performance of the solution must be within reasonable standards for the given story;

- The user interface must be developed according to the story's criteria;

- The given solution cannot have known unsolved bugs;

- Code must be reviewed by at least 1 different team member;

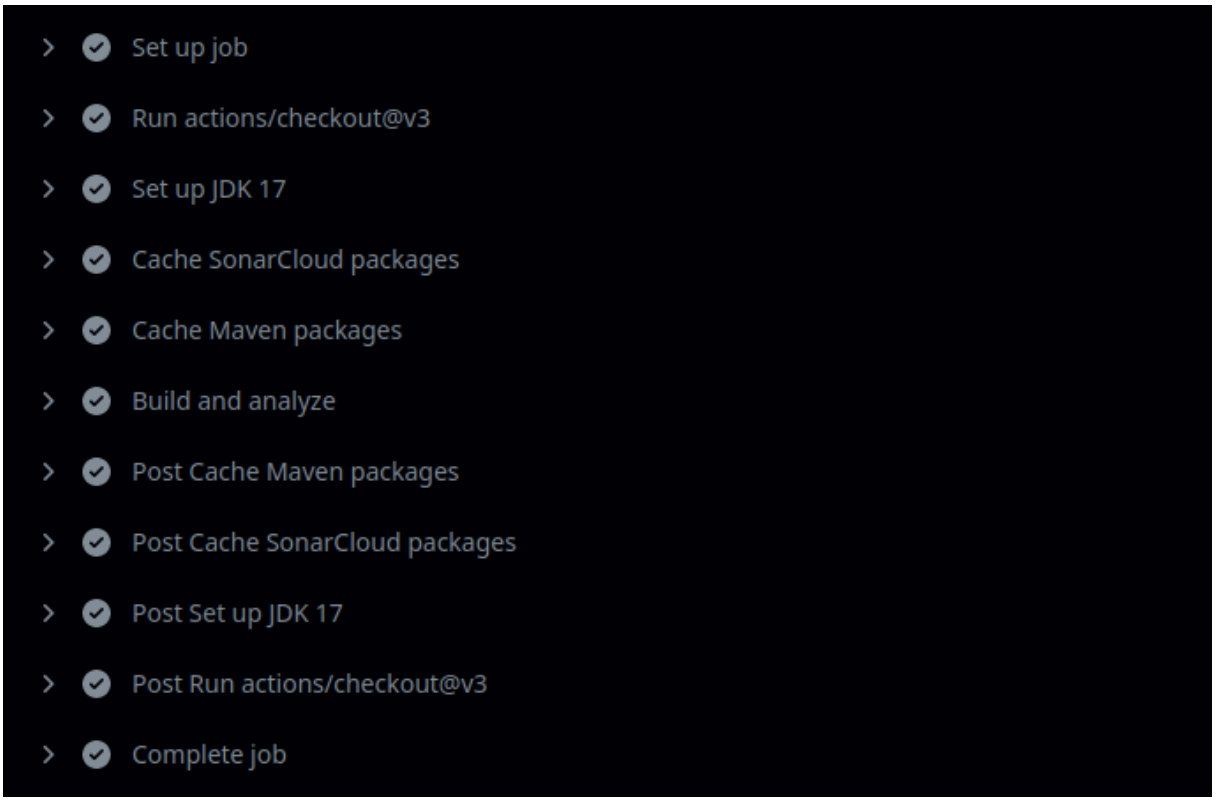
- Code must comply with the security requirements and standards of the Phihub Enterprise.

3.2 CI/CD pipeline and tools

The continuous integration and deployment pipeline is useful to ensure that updates are streamlined and that issues are found as soon as possible.
For our project, we implemented the principles of CI with the help of GitHub Actions.
With GitHub actions, the project can be tested and verified as soon as the new changes are committed into the main repository, either via regular edits or branch merges.
Since most of the project is subdivided into containers, the individual pieces of the final application are able to grow independently as long as the new changes do not break compatibility with the other services.
This containerization allows for the CD pipeline to be compartmentalized, dividing the remote Git testing to target only individual services of the codebase.

```yaml
name: SonarCloud
on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened]
```

```
      SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
    run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
```

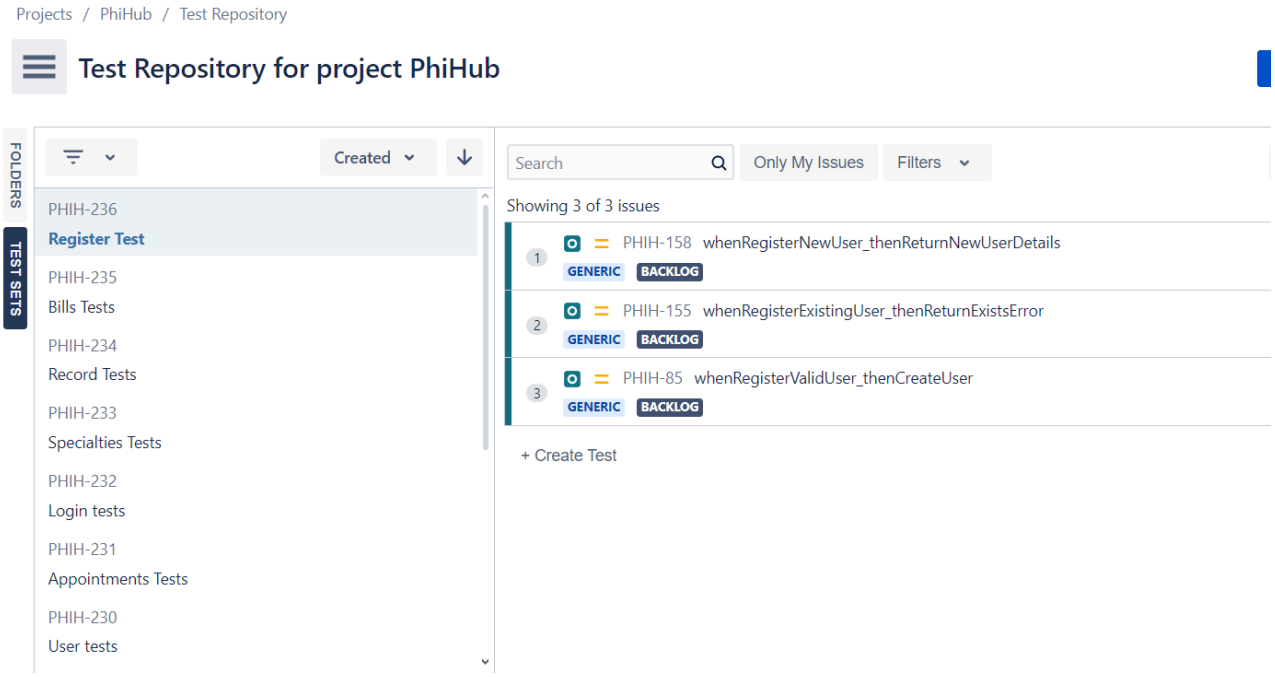


Along with automatic testing upon a pull request or push, a github action is used to upload the automatic test results to JIRA, our issue management tool.

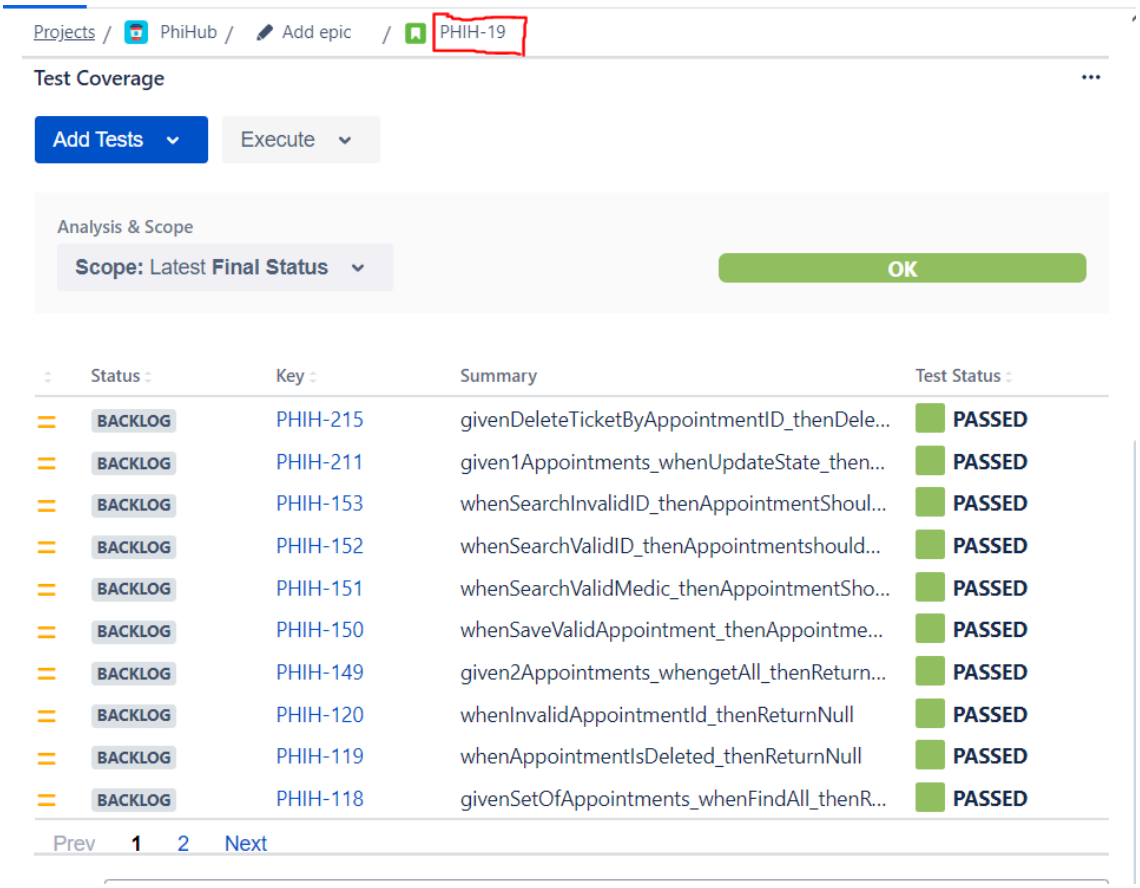
```yaml
- name: Import results to Xray
  uses: mikepenz/xray-action@v3
  with:
    username: ${{ secrets.XRAY_CLIENT_ID }}
    password: ${{ secrets.XRAY_CLIENT_SECRET }}
    testFormat: "junit"
    testPaths: "**/surefire-reports/TEST-*.xml"
    testExecKey: "PHIH-78"
    projectKey: "PHIH"
```

Through this action, we can have the given test results on JIRA, through the **XRAY** tool.

With the given test results, we created various **Test Sets,** which consist of a group of tests for the same given functionality/class.



We then linked these test sets to the respective user stories, in order to have information about the test coverage on each user story.



A github action was also developed in order to have automatic deployments.

This action combines Docker images and Github Self Hosted Runners in order to deploy the project automatically on the given Virtual Machine.

On a push to the main repository, link, this action is triggered following this process:

Each docker image is built and then pushed to Docker Hub

```yaml
build-backend-image:
  name: Push the Backend Docker image to Docker Hub
  runs-on: ubuntu-latest
  steps:
    - name: Check out the repo
      uses: actions/checkout@v4
      with:
        submodules: recursive

    - name: Log in to Docker Hub
      uses: docker/login-action@f4ef78c080cd8ba55a85445d5b36e214a81df20a
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Extract metadata (tags, labels) for Docker
      id: meta
      uses: docker/metadata-action@9ec57ed1fcdbf14dcef7dfbe97b2010124a938b7
      with:
        images: pramos16/phihub-backend

    - name: Build and push Docker image
      uses: docker/build-push-action@3b5e8027fcad23fda98b2e3ac259d8d67585f671
      with:
        context: ./Backend/
        file: ./Backend/Dockerfile
        push: true
        tags: pramos16/phihub-backend:latest
        labels: ${{ steps.meta.outputs.labels }}
```

After all images have been successfully pushed, the deployment workflow then begins.

A Github Self Hosted Runner is deployed in the machine, which is looking for jobs to complete. A guide on how self-hosted runners work can be seen here .

The final deployment github action is the following.

```yaml
deploy:
  runs-on: self-hosted
  name: "Deploy to VM"
  needs: [build-backend-image, build-signage-image, build-staff-image, build-patient-image]

  steps:
  - name: "Checkout"
    uses: actions/checkout@v4
  - name: "Deploy"
    run: |
        echo "CREATE DATABASE phihub;" > init.sql
        docker compose -f docker-compose.prod.yml pull
        docker compose -f docker-compose.prod.yml up -d --remove-orphans
        docker container prune -f
        docker image prune -af
        docker builder prune -af
```

As can be seen, it's marked with the “runs-on: self-hosted” flag which means that this action will not run in a normal github action runner, but in a self hosted one.
It first waits for all the other actions to be successful (**needs** configuration) .
After that, the action will start on the given self hosted runner, by pulling all the images that were pushed previously to Docker Hub, and then recreating the docker compose stack (docker compose up), effectively deploying the application without manual work.

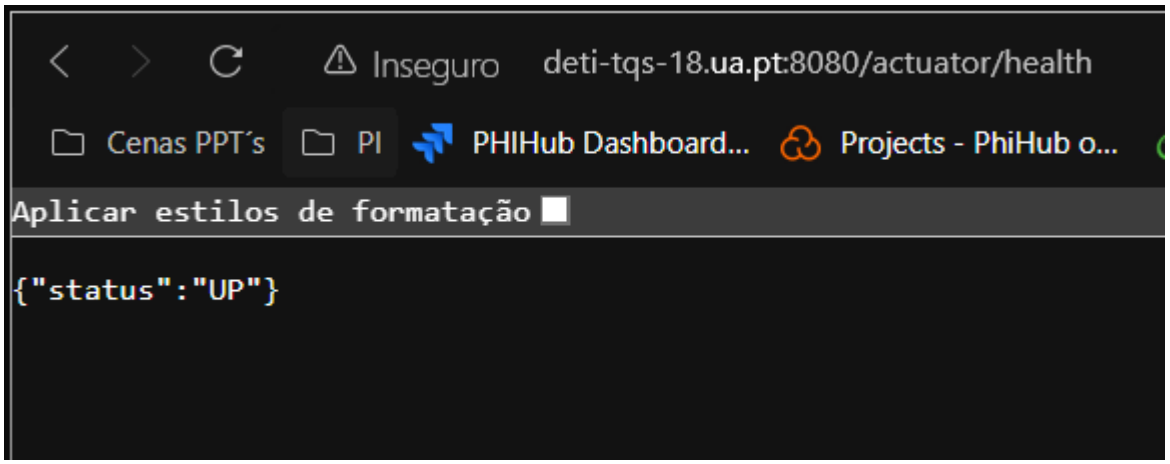
The server-side view of the github self hosted runner may be seen in the following image.

```
rodrigoaguiar96@deti-tqs-18:~/actions-runner$ ./run.sh

√ Connected to GitHub

Current runner version: '2.316.1'
2024-05-30 09:31:24Z: Listening for Jobs
2024-05-30 09:31:27Z: Running job: Deploy to VM
2024-05-30 09:32:03Z: Job Deploy to VM completed with result: Succeeded
```

With this action, we can achieve automatic deployments with the most recent changes at any given time, without having to SSH directly into the machine or set up cron jobs to deploy the application at a specific time.

3.3 System observability

Logging was implemented in the application in order to observe possible errors.
The logging driver used is **Logback,** which does logging to both the terminal and a file called backend_logs.log.
The **Spring Boot Actuator** dependency was utilized which provides various endpoints to check the application status, mainly the /actuator/health endpoint which gives us the application status.



Various attempts at implementing an ELK stack were made but the team could not get it to work correctly (various issues with connecting the logger to Logstash). Given this is an important tool for log analysis and error prevention, this would be highly considered for future work.

3.4 Artifacts repository

For the backend component, there is only 1 maven project being used, with segmentation of functionality for the different roles, patients, medics and staff and proper authentication and authorization.

Given this, there is no need for the managing of Maven artifacts, given that only 1 is used.

The given docker images which are used for deployment can be found in the following links:

Backend - https://hub.docker.com/r/pramos16/phihub-backend
Digital Signage - https://hub.docker.com/r/pramos16/phihub-signage
Staff Portal - https://hub.docker.com/r/pramos16/phihub-staff
Patient Portal - https://hub.docker.com/r/pramos16/phihub-patient

4 Software testing

4.1 Overall strategy for testing

For the overall development strategy used, it varied based on the service that was being altered.
For the backend of the application, a set of tools was used to create tests along with the development of the features, this means that the tests were mostly written at the same time as the actual feature.
Since most of the backend's components are internally separated into different sets based on the features they provide (ex: controllers, services, repositories, etc), different testing tools were used to take maximum advantage of their main feature set.
For the frontend, the tests will be written once the main functionalities of the application have been finished.
This ensures that any errors in the planned user workflows can be quickly dealt with without the need to re-write these tests.
As soon as most of the application is written, these tests will then be implemented to make sure that any additional changes do not break the final workflows used.
Once again, a mix of testing tools was used in the tests for this service, such as the use of Cucumber BDD and REST-Assured, which allows the test to simulate these user workflows.

The tests implemented make sure that almost all of the errors included in a change are found before they hit the final production environment.
The Code Quality Metrics used also allow the CI pipeline to stop any merges to the main branch if they do not comply with the Quality Gate. This is an important step in the verification of the implementation merit of the final changes into the production environment.

4.2 Functional testing/acceptance

To allow testing the complete application, as well as simulate the most common user workflows, a set of functional tests were written so that they can cover as much of the functions of the application as possible in a single test.

These tests were created using a mix of the Selenium testing framework with the Cucumber BDD tool.
The tests themselves were designed to simulate the steps that a user takes in order to perform the most common actions, such as creating an account, logging in, setting up an appointment and self-checking-in to the appointment.

During these tests, the platform was monitored in order to make sure that the correct responses were given, and that the displayed information is coherent with the input data.

More of these tests should be added as more features are created.

In our methodology for testing, these types of tests had to be created after most of the code for the frontend of platform was created, since during production, large changes could be made that would alter the workflow, making the tests fail even if the steps required for using the feature got easier to follow by the end user.

4.3 Unit tests

[Project policy for writing unit tests (open box, developer perspective) and associated resources.]

Alongside the development of features, unit tests are developed in order to ensure that the specific features added by these components work correctly.
These tests are developed using the Junit 5 Framework, with the support of the Mockito framework for mocking.

Unit tests were developed for the various layers of the Spring Boot Application (controllers, services and repositories) as well as to the created parsers and authentication layers, to ensure that they have the desired behavior even when dealing with edge cases.

4.4 System and integration testing

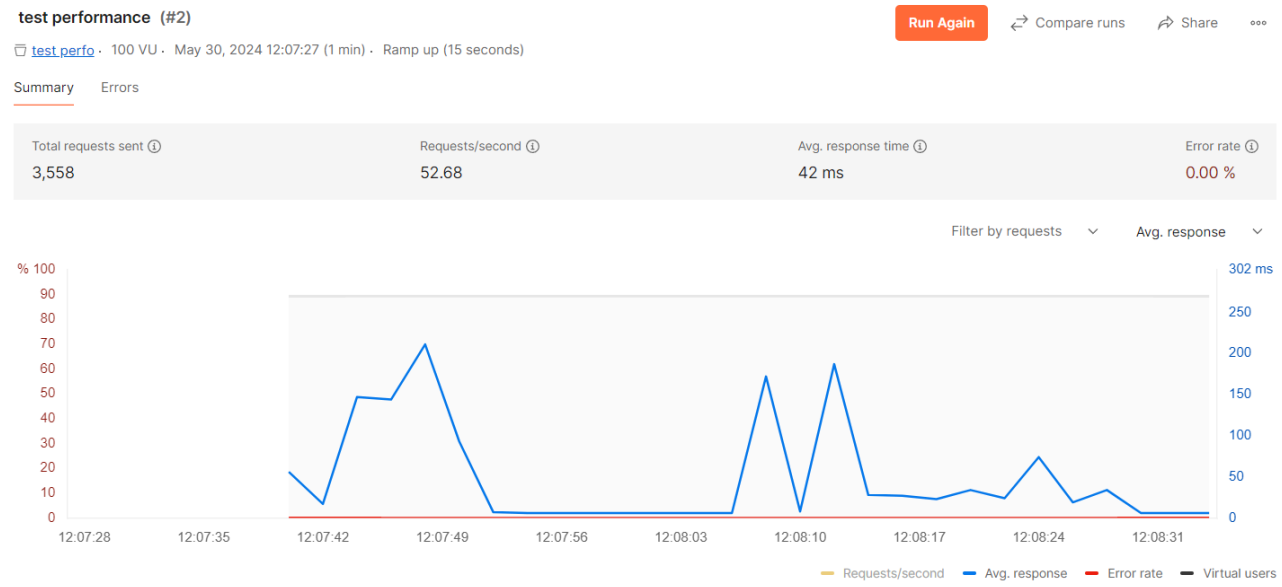
For the backend service, the integration services allow testing both the individual components and the final communication layers between them, ensuring that the whole application itself can correctly implement the features provided.

For the purpose of this test, we used an internal in-memory database.
This database was chosen for two reasons, to allow a resetting the database to be made between tests, as to not “contaminate” the database with models added by previous tests and to allow the use of pre-loaded data that can be shared between testing instances. This ensures that the same database state can be obtained at the start of every integration test.

4.5 Performance testing

Performance testing of the API was done through Postman, simulating a load of 100 virtual users sending various requests per second to the API.
More details on how the testing was performed here.

The results are the following:



Performance details for total duration

Request	Total requests	Requests/s	Resp. time (Avg. ms)	Min (ms)	Max (ms)	90th (ms)	Error %
GET get medics	3,558	52.68	42	4	1,043	46	0.00

A total of 3558 requests were sent throughout 1 minute, with an average of 52 requests per second.

The test was run locally with the application running through WSL, which means that there were other programs open on the computer and the performance is not as optimal as would be with a native Linux Machine on idle.

In general, the response time is pretty good, with an average of 42 ms, having a few peaks in the response time, most likely due to peaks in requests or other programs taking away processor time.

Still, the results are pretty acceptable which means that the API would be able to handle a relatively high load without much effect in response times or errors in the responses.