TQS: Quality Assurance manual

Gonçalo Ribau [119560], Bernardo Lázaro [119230], Rodrigo Costa [119585], Diogo Pinheiro [119907]

v2025/12/19

Contents

1 Project management

1.1 Assigned roles

We have assigned ourselves some roles we think are important to engage in a project such as this one:

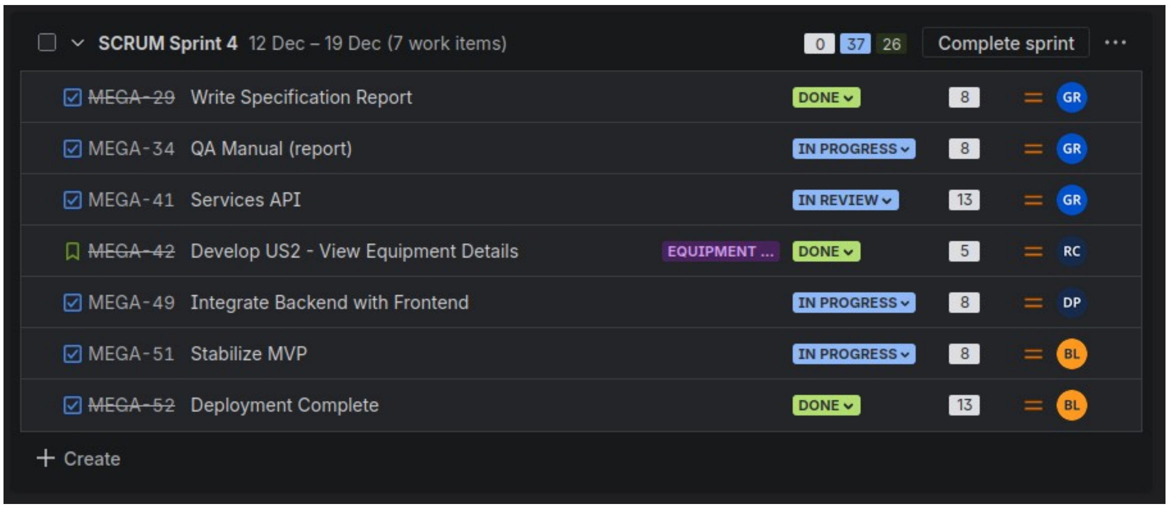
- **Team Coordinator:** Ensures that the tasks are fairly distributed and well executed. Not afraid to take matters into his own hands and assures everything is delivered on time. In our team's case, it was ***Gonçalo Ribau***.
- **Product Owner:** Represents the interests of the stakeholders. Understands the product well and should be the main guide when questions about the features arise. In our team's case, it was ***Rodrigo Costa***.

- **QA Engineer:** Responsible for articulating with the team, to promote the quality assurance practices and measure the quality of the deployment. Monitors that team follows agreed QA practices. In our team's case, it was ***Diogo Pinheiro***.
- **DevOps Master:** Responsible for the infrastructure of the development framework, and leads the preparation of the deployment's machine(s)/containers, git repository, cloud infrastructure, databases operations, etc. In our team's case, it was ***Bernardo Lázaro***.
- **Developer:** Everyone participates in the development of the system.

1.2 Backlog grooming and progress monitoring

The project follows an Agile/Scrum methodology, with sprints and tasks in each sprint.

- **Work Organization:** The requirements, in **Jira**, are created as **Tasks**, for stuff not directly related to the development itself, and **User Stories** for features. Each story is assigned a priority based on story points.

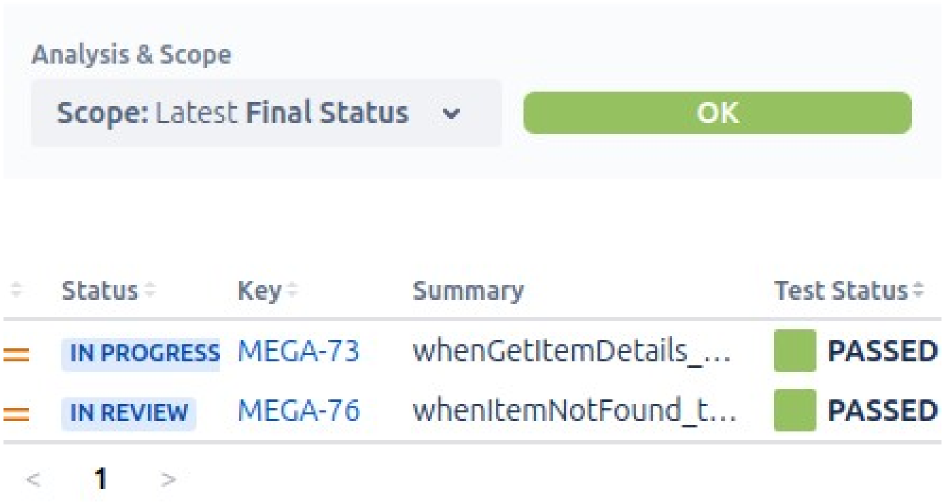


SCRUM Sprint 4 12 Dec – 19 Dec (7 work items) 0 37 26 Complete sprint

~~MEGA-29~~	Write Specification Report		DONE	8	GR
MEGA-34	QA Manual (report)		IN PROGRESS	8	GR
MEGA-41	Services API		IN REVIEW	13	GR
~~MEGA-42~~	Develop US2 - View Equipment Details	EQUIPMENT ...	DONE	5	RC
MEGA-49	Integrate Backend with Frontend		IN PROGRESS	8	DP
MEGA-51	Stabilize MVP		IN PROGRESS	8	BL
~~MEGA-52~~	Deployment Complete		DONE	13	BL

+ Create

- **Progress Tracking:** We utilize a **Kanban Board** in Jira to visualize the workflow stages (to do, in progress, in review, done). Sprint velocity is monitored using the **Burndown Chart** to ensure the team remains on schedule for the MVP delivery.
- **Requirements Coverage:** We use **Xray** for Test Management. For every user storie, there were assigned tests for automatic or manual execution, to ensure no User Storie would be complete without coverage.



Analysis & Scope

Scope: Latest Final Status — OK

Status	Key	Summary	Test Status
IN PROGRESS	MEGA-73	whenGetItemDetails_...	🟩 PASSED
IN REVIEW	MEGA-76	whenItemNotFound_t...	🟩 PASSED

< 1 >

2 Code quality management

2.1 Team policy for the use of generative AI

To maintain academic integrity while also maximizing efficiency, we opted for the use of AI assistance in some cases. For example, one of those being frontend development, as making a UI for the project wasn't one of the main concerns. The logic on the other hand, frontend and backend, being them the pillars of the project and the main product for the testing, they were developed with more caution, using only AI for generating boilerplate code, debuging complex errors, answering questions and making suggestions for cleaner code.
We made it prohibited to commit code that wasn't reviewed and understood by a member of the group, as, beyond it making us useless and unknowledgeable, AI is not perfect and most of the time, it can't code properly.
The developer who commits the code is solely responsible for it. There's no excuses of "AI wrote it!" if the code has security flaws or bugs.

2.2 Guidelines for contributors

Coding style

To ensure codebase consistency and efficient development, the team enforces some guides:

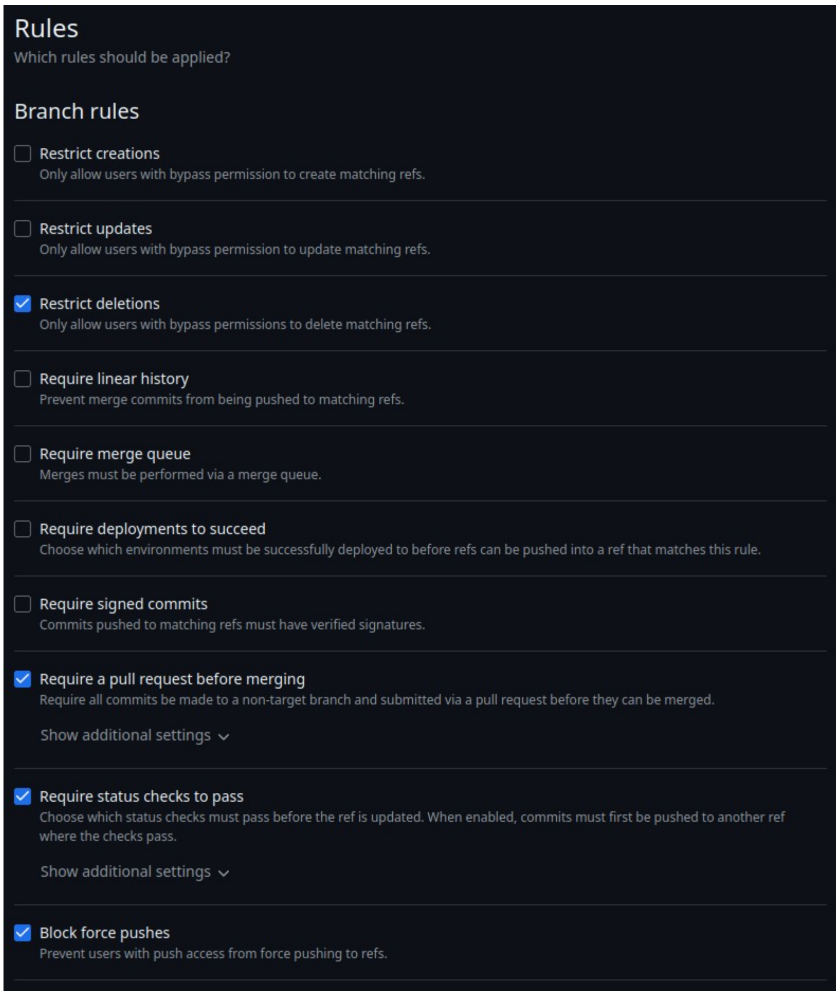
- **Backend (Java):** Adheres to standard Spring Boot conventions. We utilize **Lombok** to minimize boilerplate code (getters, setters, constructors).
- **Frontend (React):** Follows standard React functional component patterns.
- **Questions:** In case of small questions, refer to **docs** repository, or ask your colleagues.
- **Organization:** Keep the code properly organized, clean and commented.

Code reviewing

The code can't be an "If", every piece of logic in the project must be tested and reviewed:

- **Process:** No code can be merged directly into the **main** or **dev** branches. A **Pull Request** must always be opened. A **Pull Request** can not be directly merged by the developer who opened it. A different developer must review it and mark it as approved.
- **Checklist:** The reviewer must verify these conditions:
  - The logic satisfies the User Story referenced;
  - No sensitive credentials are hardcoded;
  - The CI pipeline has passed successfully.

In our project's organization, we put a **Ruleset** in the repositories to verify these conditions automatically, so there was no workaround.



Rules

Which rules should be applied?

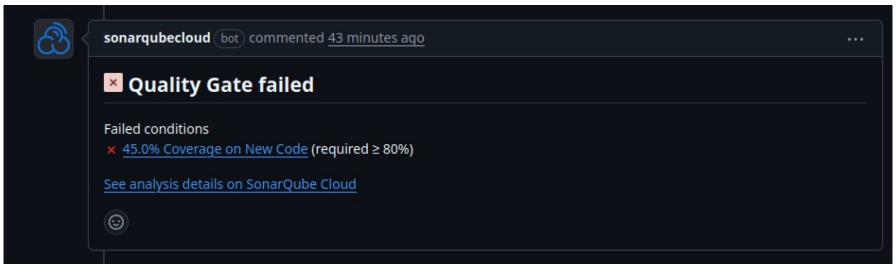
Branch rules

- [ ] Restrict creations
  Only allow users with bypass permission to create matching refs.
- [ ] Restrict updates
  Only allow users with bypass permission to update matching refs.
- [x] Restrict deletions
  Only allow users with bypass permissions to delete matching refs.
- [ ] Require linear history
  Prevent merge commits from being pushed to matching refs.
- [ ] Require merge queue
  Merges must be performed via a merge queue.
- [ ] Require deployments to succeed
  Choose which environments must be successfully deployed to before refs can be pushed into a ref that matches this rule.
- [ ] Require signed commits
  Commits pushed to matching refs must have verified signatures.
- [x] Require a pull request before merging
  Require all commits be made to a non-target branch and submitted via a pull request before they can be merged.
  Show additional settings
- [x] Require status checks to pass
  Choose which status checks must pass before the ref is updated. When enabled, commits must first be pushed to another ref where the checks pass.
  Show additional settings
- [x] Block force pushes
  Prevent users with push access from force pushing to refs.

2.3 Code quality metrics and dashboards

The project utilizes **SonarQube** for continuous inspection of code quality. It's integrated into the **CI Pipeline** to provide immediate feedback on every pull request.
The pipeline is configured to fail if the following conditions are not met:

- **Code Coverage:** Must remain above **80%** for new code;
- **Reliability:** No "Blocker" or "Critical" bugs allowed;
- **Maintainability:** Duplicated lines must be less than 3%.



sonarqubecloud bot commented 43 minutes ago

Quality Gate failed

Failed conditions
× 45.0% Coverage on New Code (required ≥ 80%)

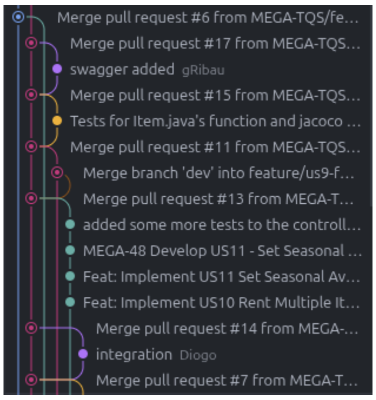
See analysis details on SonarQube Cloud

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

Our team has adopted the **Feature-Branch Workflow**, where we have the **main** branch for production, the **dev** branch for testing and **feature** branches to develop the app's **User Stories.**



```
Merge pull request #6 from MEGA-TQS/fe…
Merge pull request #17 from MEGA-TQS…
swagger added gRibau
Merge pull request #15 from MEGA-TQS…
Tests for Item.java's function and jacoco …
Merge pull request #11 from MEGA-TQS…
Merge branch 'dev' into feature/us9-f…
Merge pull request #13 from MEGA-T…
added some more tests to the controll…
MEGA-48 Develop US11 - Set Seasonal …
Feat: Implement US11 Set Seasonal Av…
Feat: Implement US10 Rent Multiple It…
Merge pull request #14 from MEGA-…
integration Diogo
Merge pull request #7 from MEGA-T…
```

Work is committed to these **feature** branches, which are, then, requested to merge with the **dev** branch, when it works well, passes the **Quality Gates** and is reviewed and approved by another developer.

When we have a stable version ready for production, the **dev** branch is requested to merge with the **main** branch, and tested and reviewed again, of course.

Definition of done

A User Story is considered "done" only when all the following criteria are met:

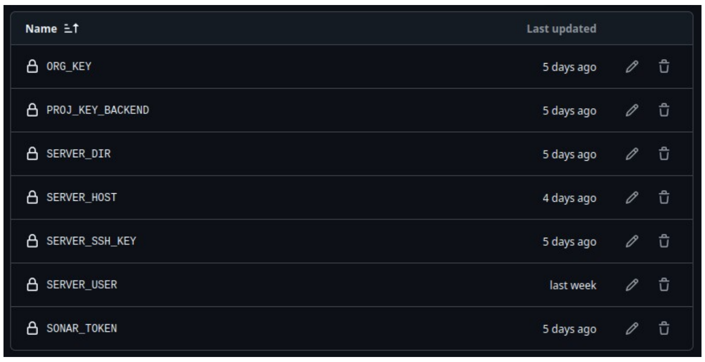
- The code is implemented and committed;
- Tests are written and passing;
- Acceptance Criteria are covered by a BDD test (Cucumber);
- Code coverage meets the 80% threshold;
- Passes all quality gates;
- The feature has been manually verified by another developer.

3.2 CI/CD pipeline and tools

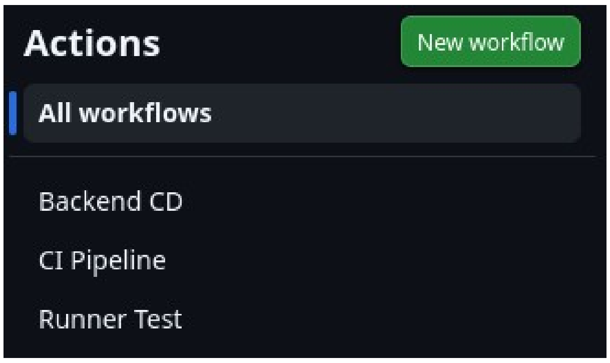
The **Continuous Integration** and **Continuous Deployment pipelines** are both orchestrated using **Github Actions**, and are dependent on **secrets**. They consist on the following automated stages:

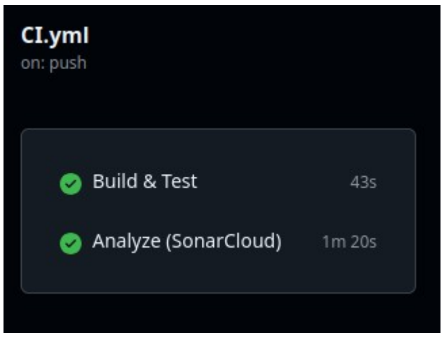
- **Checkout & Setup:** Retrieves the code and sets up the Java JDK and Node.js environments.
- **Test Execution:**
    - Runs unit and integration tests.
    - *Note:* Testing uses an in-memory **H2 Database** configuration to ensure tests are fast and isolated from the production database configuration.
- **Quality Analysis:** Runs **SonarQube** analysis and **Jacoco** coverage reports to verify Quality Gates.
- **Build & Package:** Compiles the application into a JAR file.
- **Containerization (CD):** Upon merging to **main** the pipeline builds the **Docker** images for the backend, database and frontend and prepares them for deployment.

The **Deployment Environment**, or **Production Environment**, is located on a **Virtual Machine** provided by the class, and the **Staging Environment** is simulated using **Docker** containers.



Name	Last updated
ORG_KEY	5 days ago
PROJ_KEY_BACKEND	5 days ago
SERVER_DIR	5 days ago
SERVER_HOST	4 days ago
SERVER_SSH_KEY	5 days ago
SERVER_USER	last week
SONAR_TOKEN	5 days ago





CI.yml
on: push

```
Build & Test            43s
Analyze (SonarCloud)    1m 20s
```

3.3 System observability

To ensure the system remains operational, we implemented basic observability practices:

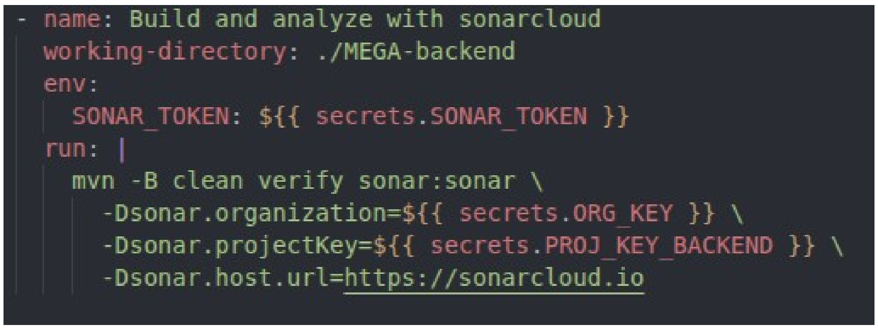
- **Logging:** All application logs (Spring Boot) and database logs (PostgreSQL) are directed to standard output (stdout), allowing them to be monitored via the Docker console.
- **Health Checks:** We utilize Spring Boot's internal mechanisms to verify that the application context is up and the database connection is active.
- **Grafana:** We implemented Grafana to get load statistics on the deployed application.

4 Continuous testing

4.1 Overall testing strategy

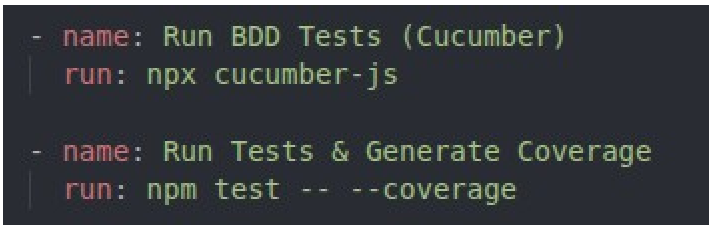
The project follows the **Testing Pyramid Strategy** to ensure a robust and maintainable test suite:

- **Unit Tests (70%):** Validates isolated business logic (Services and Utility classes) using mocks. These are fast and run on every commit.
- **Integration Tests (20%):** Validates the interaction between components (Controllers, Services, Repositories) and the database.
- **E2E/BDD Tests (10%):** Validates full user flows from the user's perspective. These are slower and focus on critical paths.



```
- name: Build and analyze with sonarcloud
  working-directory: ./MEGA-backend
  env:
    SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
  run: |
    mvn -B clean verify sonar:sonar \
      -Dsonar.organization=${{ secrets.ORG_KEY }} \
      -Dsonar.projectKey=${{ secrets.PROJ_KEY_BACKEND }} \
      -Dsonar.host.url=https://sonarcloud.io
```

In the backend's pipeline, we run "*mvn verify*" that runs unit tests and integration tests.



```
- name: Run BDD Tests (Cucumber)
  run: npx cucumber-js

- name: Run Tests & Generate Coverage
  run: npm test -- --coverage
```

In the frontend's pipeline, we run "*npx cucumber-js*" that runs the E2E/BDD tests, and "*npm test*" that runs normal *jsx* tests.

4.2 Acceptance testing and ATDD

We employ **Acceptance Test-Driven Development (ATDD)** using **Cucumber**, in the frontend repository. Before implementation begins, the developer and Product Owner define the Acceptance Criteria in Gherkin syntax (`.feature` files). These scenarios serve as executable documentation, and the developer should develop these so he ensures the system behaves exactly as specified in the requirements.

4.3 Developer facing tests (unit, integration)

Unit Testing:

- Developers are required to write unit tests for all service-layer logic, mocking external dependencies (like repositories) to test in isolation. These tests make up 70% of all the tests for the new code developed. In our project we have implemented them using **JUnit 5** and **Mockito**.

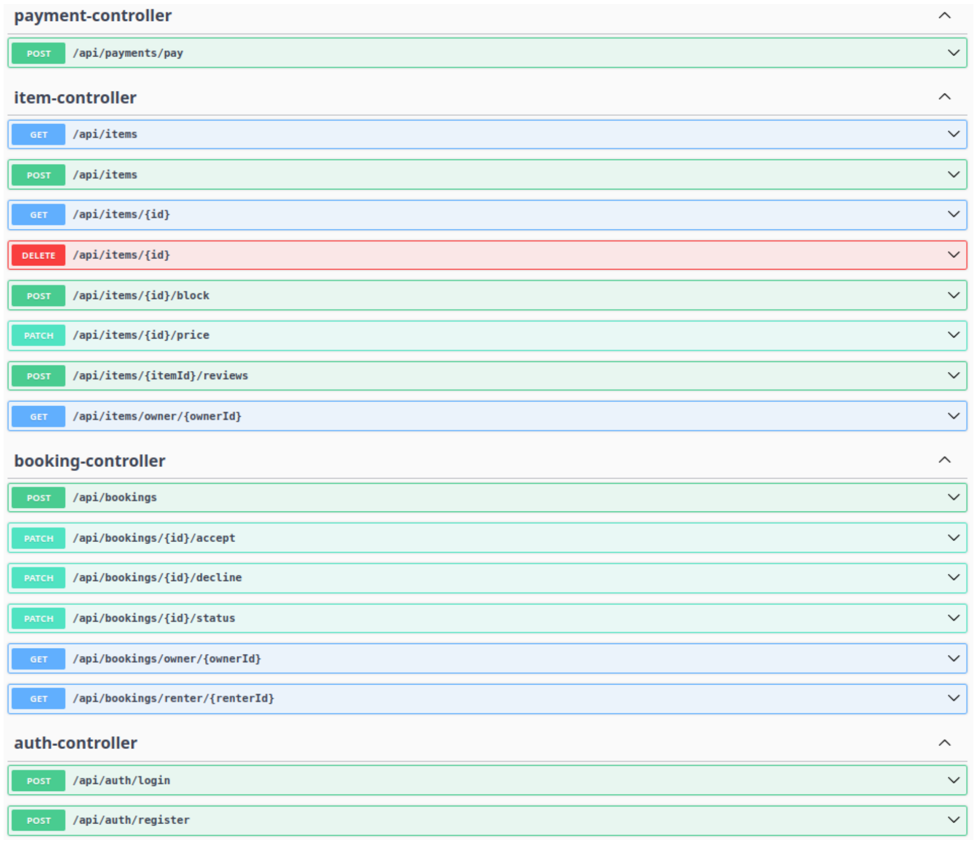
Our most relevant unit tests consist of the ones for the service and data layer, more specifically for the ItemService and ItemRepository, which test the logic in the manipulation of the items and their information.

Integration Testing:

- To ensure stability and speed in the CI environment, these tests run against an **H2 in-memory database**, while the production application runs against **PostgreSQL**. This separation ensures that tests do not require a live Docker environment to run. They make up 20% of all the tests for the new code developed. Our Integration Tests are implemented using Spring Boot Test.

4.4 Exploratory testing

We use the auto-generated documentation from Swagger (http://deti-tqs-17.ua.pt:8080/docs) for unscripted testing. We set up the database and the backend, and send requests using Swagger to test some edge cases, error messages, and JSON formats, without need for frontend.



payment-controller

POST	/api/payments/pay

item-controller

GET	/api/items
POST	/api/items
GET	/api/items/{id}
DELETE	/api/items/{id}
POST	/api/items/{id}/block
PATCH	/api/items/{id}/price
POST	/api/items/{itemId}/reviews
GET	/api/items/owner/{ownerId}

booking-controller

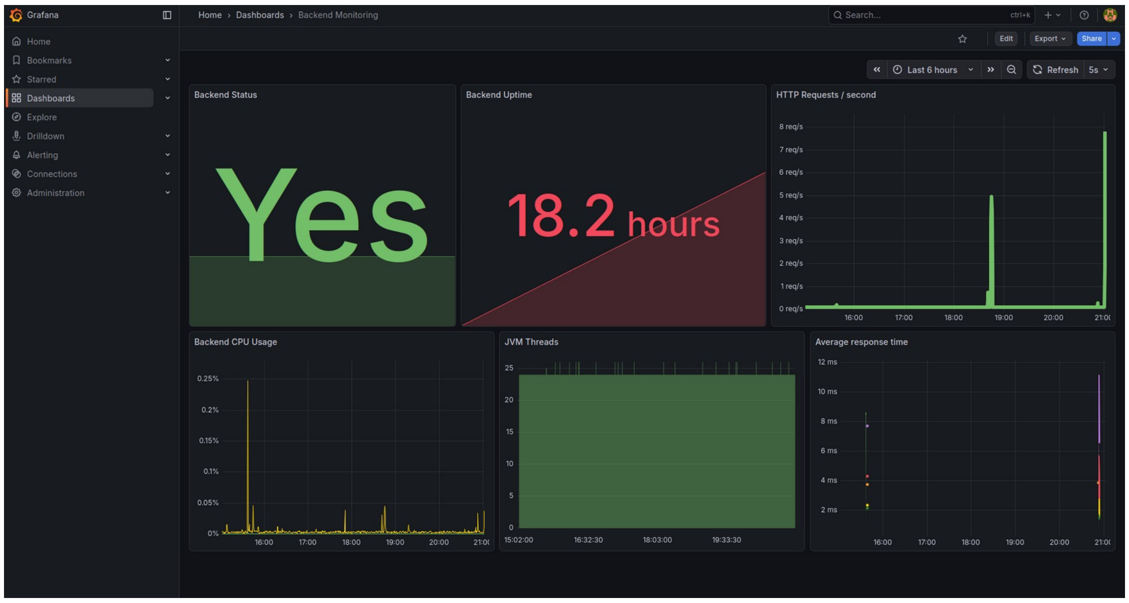
POST	/api/bookings
PATCH	/api/bookings/{id}/accept
PATCH	/api/bookings/{id}/decline
PATCH	/api/bookings/{id}/status
GET	/api/bookings/owner/{ownerId}
GET	/api/bookings/renter/{renterId}

auth-controller

POST	/api/auth/login
POST	/api/auth/register

4.5 Non-function and architecture attributes testing

We utilize **k6** to perform load testing on critical endpoints (for example, search and booking creation). This ensures the system maintains acceptable response times even under concurrent user load.


Grafana
Home › Dashboards › Backend Monitoring
Home
Bookmarks
Starred
Dashboards
Explore
Drilldown
Alerting
Connections
Administration
Search...
Edit
Export
Share
Last 6 hours
Refresh
5s
Backend Status
Yes
Backend Uptime
18.2 hours
HTTP Requests / second
Backend CPU Usage
JVM Threads
Average response time