TQS: Quality Assurance manual

Filipe Viseu [119192], Duarte Branco [119253], Samuel Vinhas [119405], André Ferreira [119061]
v2025-12-14

Contents

1 Project management

1.1 Assigned roles

- **Team Coordinator (a.k.a. Team Leader)** - André
- **Product owner** - Samuel
- **QA Engineer** - Filipe
- **DevOps master** - Duarte

1.2 Backlog grooming and progress monitoring

We adopted Agile methodologies, specifically Scrum, utilizing Jira (see our Jira space) as our primary project management tool. The backlog is organised by Epics (ex.: Booking & Scheduling), which are broken down into User Stories (ex.: US2 - Book an item). Each Sprint takes 1 weak and, before each one, the team refines the backlog, clarifying the acceptance criteria and assigning priorities. We also discussed and used Story Points based on the task's amount of work/complexity it required.

2 Code quality management

2.1 Team policy for the use of generative AI

In our team, AI (GitHub Copilot, ChatGPT, etc.) is allowed and encouraged simply to make the process easier, but we do have some rules and caveats.

Do's:

- Use AI tools (GitHub Copilot, ChatGPT, etc.) to accelerate development.
- Generate edge-case scenarios and mock data for unit tests.
- Leverage GitHub Copilot for initial PR reviews.

Don'ts:

- Never copy-paste AI-generated code without understanding its logic.
- Never commit code you cannot explain to another team member.
- Do not rely solely on AI reviews; human review is mandatory.

Golden Rule: "If you cannot explain how the code works to another person, you are not allowed to commit it".

2.2 Guidelines for contributors

Coding style

Backend (Java):

- Google Java Style Guide for all Java code.
- Class names in PascalCase.
- Method and variable names in camelCase.

- All code, comments, and commit messages in English.
- No unused imports.

Frontend (JavaScript/React):

- Airbnb JavaScript Style Guide
- Prefer functional components and hooks.
- File and component names in PascalCase.
- Single quotes for strings.
- Always use semicolons.

Code reviewing

Every Pull Request requires **at least two of the team member's approval** before merging to the main branch.

Checklist:

- ☑ Does the code satisfy the User Story acceptance criteria?
- ☑ Are there new tests covering the changes?
- ☑ Is the code readable and maintainable?

We used GitHub Copilot to perform an initial verification in some PRs, checking for syntax errors, typos, and simple logic flaws, allowing our team to focus on architectural logic.

2.3 Code quality metrics and dashboards

We used SonarCloud (see latest review) to maintain code quality. In our system, the pipeline will fail if the code does not meet the following criteria:

- ☑ Coverage: New code must have >= 80% test coverage.
- ☑ Reliability: 0 Bugs.
- ☑ Security: 0 Vulnerabilities.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

We used the Feature Branch Workflow:

1. **Pick a Task** - The developer moves a User Story to "In Progress" in Jira.
2. **Create new branch** - In Jira, there's an option to create a new branch from main named after the issue (ex.: feature-US1-search-consoles).
3. **Push & PR** - Push the new changes and open a new Pull Request targeting the main branch.
4. **CI Checks** - GitHub Actions triggers the build, tests, and Sonar analysis.
5. **Review & Merge** - Once CI passes and two members approve, the code is merged to the main branch.

Definition of done

In our system, for a User Story to be considered "Done", it needs to complete this 5 checks:

- ☑ Code is implemented and strictly follows coding standards.
- ☑ Unit Tests (JUnit) and Integration Tests are written and passing.
- ☑ Acceptance Tests (Cucumber/BDD) match the User Story criteria and pass.
- ☑ Static Analysis (Sonar) passes the Quality Gate.
- ☑ Peer Code Review is completed and approved.

3.2 CI/CD pipeline and tools

We use GitHub Actions to orchestrate our pipeline. We made 2 files for this - build.yml and deploy.yml. The first one builds, tests and verifies everything is like it should, and it triggers on any branch, so, the feature branches are included. The second takes care of the deployment process to the VM, so, it runs a self-hosted runner (on the VM) that starts all our containers, and it only triggers on the main branch, so there's no problem with deploying unreviewed code.

Build Pipeline (trigger on any branch):

1. **Checkout** - Fetch repository with full history.
2. **Setup** - Configure JDK 21 with Maven caching.
3. **Wait for Services** - Ensure PostgreSQL is ready.
4. **Build & Test** - `mvn verify` runs all tests (unit, integration, BDD).
5. **SonarCloud Analysis** - Static code analysis and quality gate check.

Assuming the PR was merged and pushed into main we can now deploy.

Deploy Pipeline (trigger on main):

1. **Checkout** - Fetch repository with full history.
2. **Create .env file** - it uses GitHub secrets to get the necessary .env file for the project to run.
3. **Restart Docker Containers** - with the new code.

3.3 System observability

To ensure proactive monitoring of the system's operational conditions, a centralized monitoring stack was deployed on the production virtual machine using **Prometheus** and **Grafana**. This setup enables continuous, real-time observation of both the **VMs infrastructure** and the **app/container metrics**, allowing early detection of any performance issues or abnormal runtime behavior.

We used 2 different dashboards:

- **Node Exporter Full**
- **Cadvisor exporter**

This is the data continuously collected and stored:

Infrastructure metrics (Node Exporter)

- CPU utilization (per core and total).
- Memory usage (used, free, cached).
- Disk usage and filesystem availability.
- Network throughput and packet rates.
- System load averages.

Container and application metrics (cAdvisor)

- CPU usage per container.
- Memory consumption per container.

- Container restarts and uptime.
- Network I/O per container.
- Disk I/O per container.

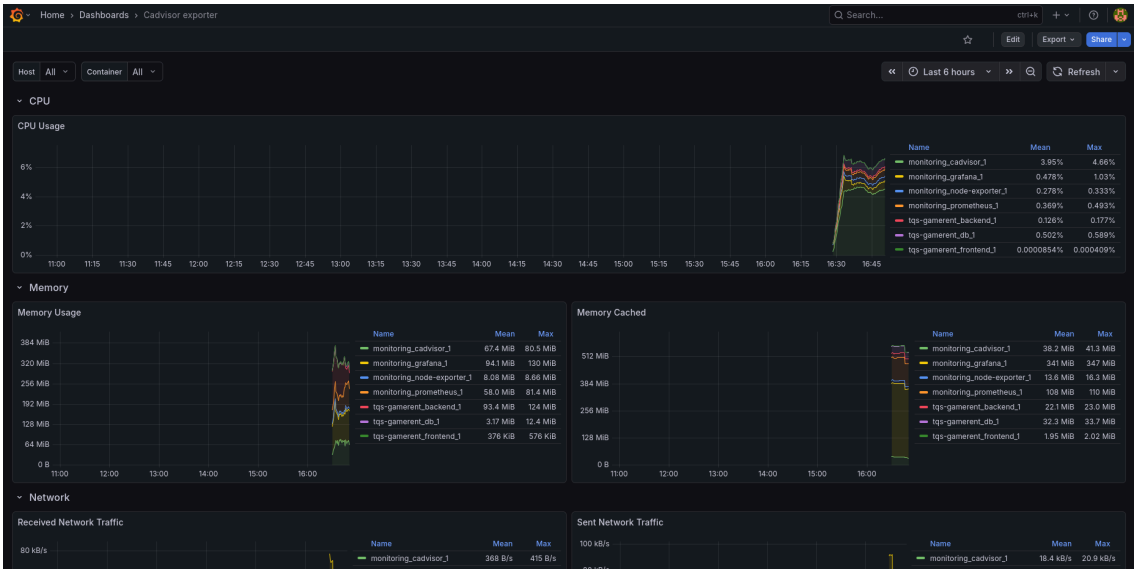

Fig 1: Node Exporter Full dashboard of our VM

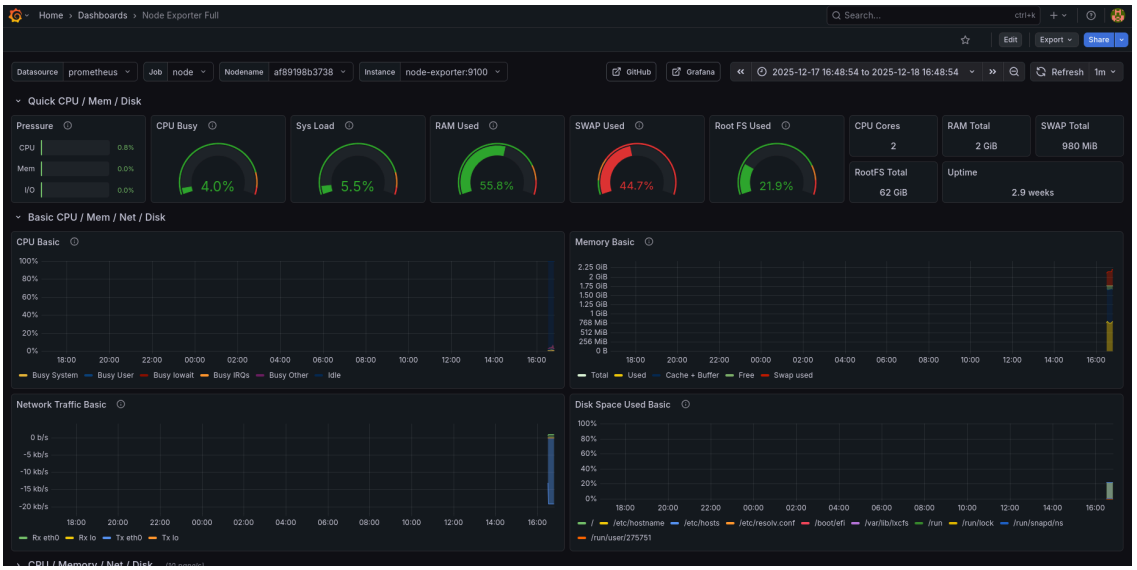

Fig 2: Cadvisor exporter dashboard of our VM

Although no automated alerting rules were configured at this stage, this setup already allows for visual and threshold-based detection of critical events, which we think is enough for this project. But, we might improve and do automated alerts in the future.

4 Continuous testing

4.1 Overall testing strategy

Our testing strategy follows the **Testing Pyramid**, aligned with our CI/CD process.

Unit Testing (TDD):
Tools: JUnit 5 (Backend)
Practice: Developers write tests before implementation.

Behaviour Driven Development (BDD):
Tools: Cucumber.
Practice: Feature files are written in Gherkin syntax (Given/When/Then) derived directly from the User Stories (e.g., US1: Search functionality).

End-to-End (E2E) Testing:
Tools: Playwright.
Practice: Simulates real user scenarios (e.g., "Miguel books a PS5").

Performance Testing:
Tools: K6.
Practice: Scripts simulate high traffic to ensure the system handles load, stress and spike.

4.2 Acceptance testing and ATDD

All acceptance tests must be written in **Gherkin syntax** (Given/When/Then) using Cucumber.

Policy:

- We must define the .feature files before implementation begins.
- They are readable by every “type” of stakeholder (technical and non-technical).
- This clarifies requirements and prevents "building the wrong thing".
- The tests interact with the application through the UI (using Playwright) or the public API, verifying that a user's “journey” completes successfully.

4.3 Developer facing tests (unit, integration)

Unit tests
Strategy - We adopt a **TDD (Test Driven Development)** approach:

1. **Red**: Write a failing test for a specific logic.
2. **Green**: Write just enough code to pass the test.
3. **Refactor**: Clean up the code while ensuring tests still pass.

We tested everything (logic and API's) and when we needed to mock, we used **Mockito**. All tests were written before the production code, aiming for high coverage in business logic classes.
Most Relevant Unit Tests:

- **Service Layer Logic**: e.g., RentalServiceTest (verifying that a renter cannot book an item if it is already booked for those dates).
- **Validation Logic**: e.g., Verifying that a User cannot register with an invalid email format.
- **Utility Methods**: e.g., Date calculation utilities (calculating the difference between pickup and drop-off dates).

As for Integration Tests, we will use MockMVC while auto-wiring other classes that we need.

API testing is a subset of our integration strategy.
Since our system relies on a React frontend communicating with a Java backend, the API is crucial.
For that, we need to make sure endpoints return 200 for success, 400 for bad requests, 401 for unauthorized access, etc.
Besides that, we need to verify the JSON response matches the expected format (example: The search results from US1 contain specific fields like name, platform, price).
Tooling: We utilize MockMvc within our JUnit/Spring integration tests to send HTTP requests and validate responses without launching the full server.

4.4 Exploratory testing

Before we merge a pull request, the 2 members that will give their review must act as an User that wants to break the system, for example, an user trying to change details on an item that he doesn't own. Usually we did manual tests for like 20-25 minutes but that depended on the type of PR, in the admin ones we took more time because it was crucial.

4.5 Non-function and architecture attributes testing

Performance: We used K6 to simulate load, stress and spike.
Load Test: Simulate 50 concurrent users to ensure response time < 200ms.
Stress Test: Find the breaking point of the API.
Spike Test: From 10 users to 100 in 10 seconds while ensuring that all responses still take less than 1 second

Architecture: We use ArchUnit to enforce architectural constraints (ex.: "Controllers should not access Repositories directly").