TQS: Quality Assurance manual

Miguel Alexandre Moura Neto [119302]
Alexandre Filipe da Paula Andrade [119279]
Thiago Aurélio Pires Barbosa Vicente [121497]
João Pedro Silva Pereira [120010]

v2025-12-16

Contents

1 Project management

1.1 Assigned roles

Team Coordinator: Alexandre Andrade

Product owner : Miguel Neto

QA Engineer: João Pereira

DevOps master : Thiago Vicente

1.2 Backlog grooming and progress monitoring

We use JIRA as our main organization and work-tracking tool.

Our team follows agile project management practices, working incrementally through user stories managed in time-boxed sprints. The backlog is continuously refined in JIRA, where user stories are reviewed, prioritized, and updated as requirements evolve.

Progress is tracked by assigning story points to each user story and organizing the work into 1-week sprints. During the sprint, task status is monitored directly in JIRA to ensure alignment with planned timelines. Sprint progress is reviewed regularly to confirm that work is proceeding according to schedule.

We hold weekly meetings where:

- The previous sprint is reviewed, including completed and pending user stories
- Any blockers or scope adjustments are discussed
- Upcoming sprints are planned in advance, with story prioritization and point estimation updated in JIRA

This routine allows the team to proactively monitor progress, adapt to changes early, and maintain a consistent delivery cadence.

This is our Jira project: Jira Project

2 Code quality management

2.1 Team policy for the use of generative AI

AI-assistants might be used to generate code for production , as it will be tested.
For the tests, AI-assistants may be used, but they have to be carefully inspected by a developer, because the tests ensure that the code is working and that it has quality, so they need to accomplish their objective. The code production can have more AI generated code, because the tests will ensure that it works.

2.2 Guidelines for contributors

Coding style

Our coding style follows Spring Boot conventions with Java practices, emphasizing consistency above all else. When editing existing code, we first examine the surrounding patterns to match local style preferences, ensuring new code doesn't disrupt the reading rhythm. We use explicit imports rather than wildcard imports, maintain four-space indentation with eight-space continuation indents, and keep methods focused when possible - mostly breaking up methods that exceed 40 lines unless it harms program structure.

Exception handling is principled, never ignoring exceptions with empty catch blocks without explicit justification and avoiding generic exception catches except in specific top-level or test code scenarios.

Code reviewing

Code reviews are done by the peers. Copilot may be used to help with the reviews but ideally he shouldn't be the only reviewer of a Pull Request. All pull requests will be evaluated by the Continuous Integration pipeline, and if they don't pass either the tests or the SonarQube quality gates, they can't be merged until the issue is fixed.

Another developer then reviews the code and can either approve or disapprove. If approved, the pull request may then be merged into the correct branch (usually dev or main).

2.3 Code quality metrics and dashboards

For code quality metrics, we are using SonarQube. As the quality gate we'll be using the default from sonar.
This is our SonarQube Dashboard: SonarQube Dashboard

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

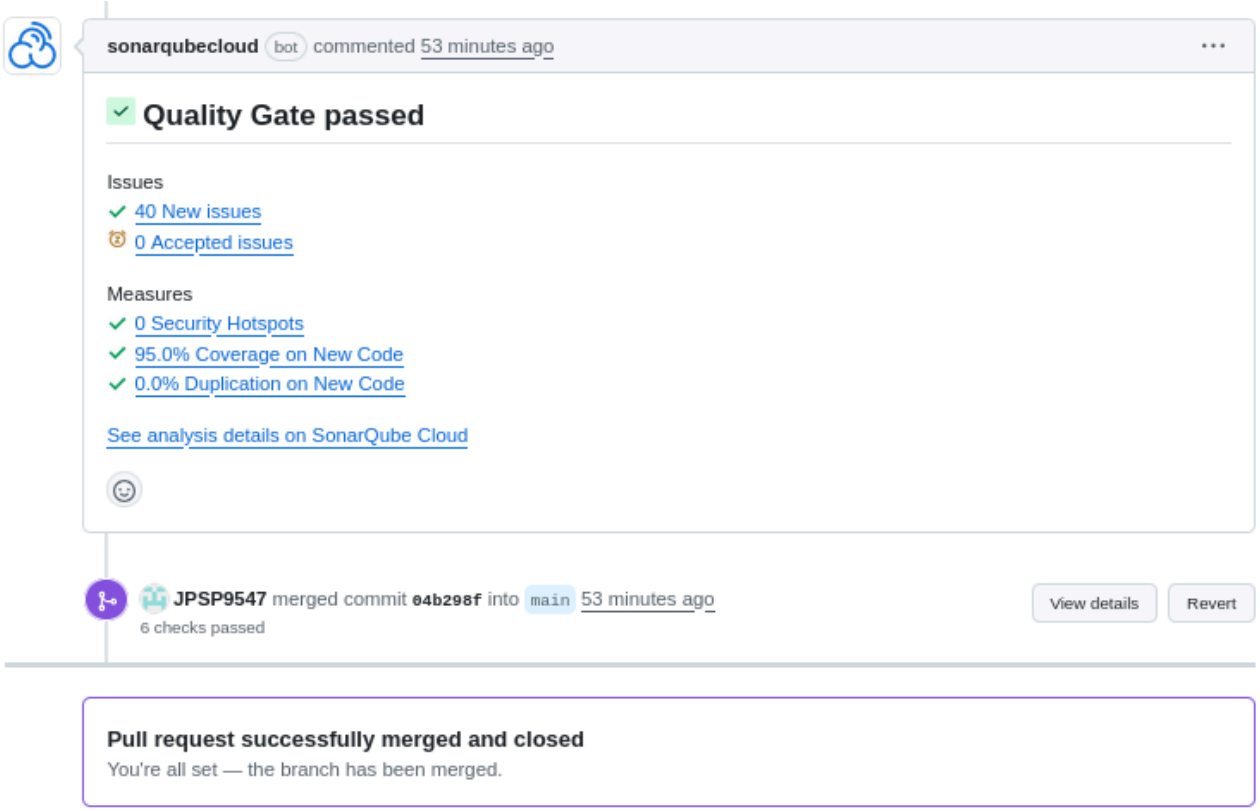
First of all, we choose the tasks/user stories for the next sprint, after that we assign each task/user story to one developer.
We'ill use the GitFlow workflow.
Each user story will have a feature created on the git repository and after it is done, a pull-request will be done and after a review it will (or not) be merged into the dev branch. When it's ready for production it will be put on the main branch.
When a commit is pushed onto github a SonarQube code analysis is triggered and the results are displayed on the dashboard.

Definition of done

A user story is considered done, when its features are done and it passes all the acceptance criteria defined for it, passes the quality gate and is merged into the dev branch. It is expected to have tests mapping to each acceptance criteria, thus it should pass all those tests.

3.2 CI/CD pipeline and tools

For continuous integration, we have github actions running tests and SonarQube analysis on each new code that is committed, that way we can address bugs and errors early, before they cause damage. Each commit is also automatically analysed by SonarQube to check for code quality.
The analysis also run on any pull request that is created, and the pull request can be merged only if the tests and the SonarQube analysis both pass.
For continuous delivery, it is based on Docker Containers. It deploys a new version each time that new code is pushed into the main branch of the repository, and only deploys if that code passes the tests and quality gate.



3.3 System observability

Monitoring of the system is done continuously collecting the logs that the system generates while operating and showing them to the administrator on a web page. The logs have different categories, it might be “INFO”, “WARN” or “ERROR”. This monitoring page also shows statistics relating to the Database (current connections) and the number of requests the system is taking along with the time the system takes to respond.
JVM related statistics (like Memory consumption) should also be shown.

4 Continuous testing

4.1 Overall testing strategy

For testing, we will be doing TDD to better align with CI (first we make the tests and they don't pass, and only after the code is done the tests will start to pass). Each component that is created must have unit tests. If a component is modified and a new feature is added, unit tests should be modified to account for that, but if a component is modified, and the modifications just changed the implementation and didn't add new features, the current tests are enough.
BDD will be used to test the backend API and will also be used with Selenium to test the UI and its integration with the backend.

Evidence of CI Integration

- Tests must pass before merge (PR gating)
- Coverage reports are generated and analyzed automatically
- SonarCloud enforces quality gates (code smells, coverage thresholds, security vulnerabilities)
- Deployment only proceeds after successful test execution

4.2 Acceptance testing and ATDD

For writing acceptance tests, we will be using BDD with Cucumber and using Selenium for UI. The acceptance tests will test the behavior of the system in a user-facing perspective, through the UI and also use a black-box approach since it won't care about the system architecture, just the behavior.
The developer needs to write these each time a user story is completed, or code changes to add a new feature, or fixing a bug that affects user experience.

Acceptance tests are written in Gherkin (.feature files), implementing Acceptance Test-Driven Development (ATDD). Each feature file maps to a user story.

Feature Coverage (13 features):

Feature File User Story Coverage
secure_payment.feature OLS-38: Renter pays securely for rental
search_instruments.feature Search instruments by name
search_music_sheets.feature Search music sheets
filter_instruments_by_type.feature Filter instruments by type
filter_sheets_by_category.feature Filter sheets by category
register_instrument.feature Register new instrument
register_music_sheet.feature Register new music sheet
view_instrument_details.feature View instrument details
view_music_sheet_details.feature View music sheet details
manage_availability.feature Manage instrument availability
review_items.feature Review rented items
review_renters.feature Review renters
admin_oversight.feature Admin oversight capabilities

4.3 Developer facing tests (unit, integration)

Unit tests should be open box and developer perspective, as they should validate classes and internal logic while isolated. We will use Junit with Assertj for unit testing. Unit tests shall be written for each class that has no trivial code (ex: only getters and setters don't need to be tested, when code is refactored unit tests need to be modified to take that into account, a new feature is added on a class. The most relevant unit tests are the ones that cover the business logic, as they validate how the system core works.

```java
@ExtendWith(MockitoExtension.class)
class BookingServiceTest {

    @Mock
    private BookingRepository bookingRepository;

    @Mock
    private ItemRepository itemRepository;

    @InjectMocks
    private BookingService bookingService;

    @Test
    void whenApproveBookingByOwner_thenStatusIsApproved() {
        when(bookingRepository.findById(1L)).thenReturn(Optional.of(booking));
        when(bookingRepository.save(any(Booking.class))).thenReturn(booking);

        Booking approved = bookingService.approveBooking(1L, 10);

        assertThat(approved.getStatus()).isEqualTo(BookingStatus.APPROVED);
        verify(bookingRepository, times(1)).save(booking);
    }

    @Test
    void whenApproveBookingByNonOwner_thenThrowException() {
        when(bookingRepository.findById(1L)).thenReturn(Optional.of(booking));

        assertThatThrownBy(() -> bookingService.approveBooking(1L, 999))
            .isInstanceOf(IllegalArgumentException.class)
            .hasMessageContaining("not authorized");
    }
```

For integration tests, we will use a closed box developer perspective, but for some Spring Boot integration tests, they might be open-box (ex: testing how the repository interacts with the DB). Integration tests shall be written when the API changes, when the connection between layers changes, when one layer changes its interfaces or when an external service is implemented.

4.4 Exploratory testing

Exploratory testing will be done more on the frontend part of the system to ensure the system is working as it should from the user perspective.
All the user stories will be tested in a real production environment.

4.5 Non-function and architecture attributes testing

Non functional testing will also be done to simulate and test how the backend works under load using K6 and testing the frontend load times and accessibility using LightHouse.

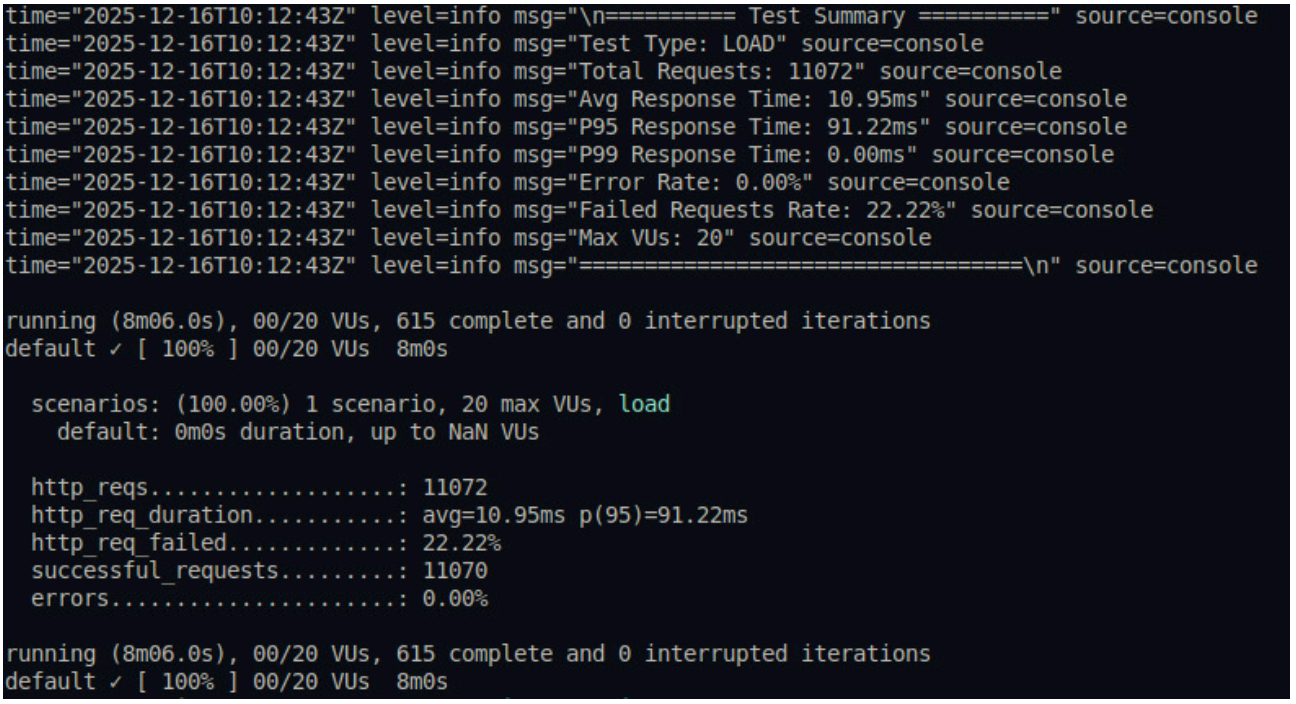
For backend load testing with K6, three types of tests will be run, each serving a different purpose:

Smoke tests act as a quick sanity check before running more intensive tests. These use only 1-5 virtual users for a very short duration to verify the basics: does the app start, do critical endpoints respond, does authentication work? Running these fast tests in CI pipelines provides immediate feedback on whether the deployment is ready before investing time in heavier testing.

Load tests measure how the application behaves under realistic, sustained traffic. These tests ramp up to a target number of users and maintain that load for several minutes to observe steady-state performance. This reveals average and tail latencies (especially p95 and p99 percentiles), resource usage patterns, and whether the service degrades over time. The goal is to find the maximum sustainable load where the system still maintains acceptable latency and error rates, and to catch slow accumulative issues like memory leaks or connection pool exhaustion.

Spike tests stress the system with sudden traffic bursts to evaluate resilience and autoscaling. These rapidly ramp from low to high user counts, hold briefly, then ramp back down. This shows how the system handles unexpected surges—whether requests queue properly, if timeouts occur, how error rates spike, and how quickly the system recovers. The aim is to ensure graceful degradation and confirm that infrastructure components like autoscalers and database connection pools can tolerate bursts.

The following image represents the result of a load test:



The test ran for 8 minutes total, ran 615 complete iterations (each user completed their full test scenario multiple times), ramped up to 20 concurrent users and made 11,072 HTTP requests in total

Results Summary:
The system performed well under this load level:

- Latency: Average response time was 10.95ms with a p95 (95th percentile) of 91.22ms
- Throughput: Maintained approximately 22.22 requests per second
- Reliability: Achieved 0.00% error rate with 11,070 successful requests out of 11,072 total

- Stability: All 615 iterations completed without interruption, indicating the system remained stable throughout the test

  At 20 concurrent users, the backend handled the sustained load effectively with consistently low latency (sub-100ms for 95% of requests), zero errors, and no signs of degradation over the 8-minute period. This suggests the system has headroom for additional load.

From the whole k6 test set, we got the following results:

Test Type	Duration	Max VUs	Requests	Avg Response	P95 Response	Error Rate	http_req_failed
Smoke	30s	1	56	17.29ms	102.25ms	0%	21.43%
Load	8m	20	11,072	10.95ms	91.22ms	0%	22.22%
Spike	4m	50	9,65	16.33ms	101.17ms	0%	22.22%