deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

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

Since the group consisted of four members, each one took one of the “main roles”, as follows:

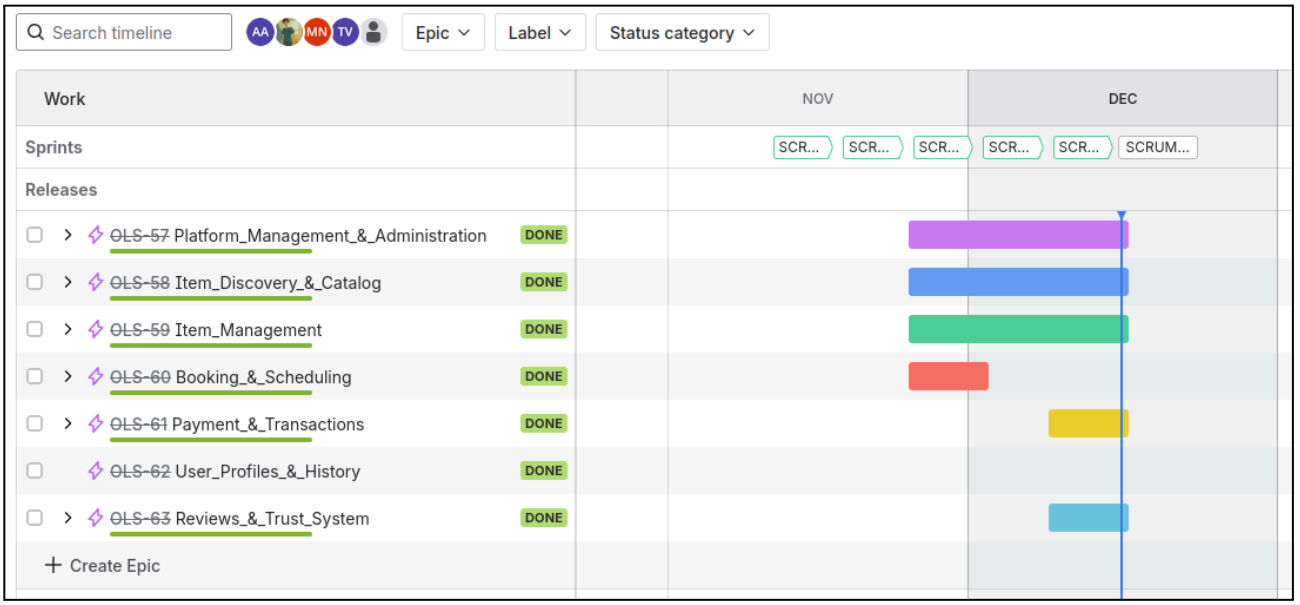
- Team Coordinator: Alexandre Andrade
- Product owner : Miguel Neto
- QA Engineer: João Pereira
- DevOps master : Thiago Vicente

All the members above also played the role of developer.

1.2 Backlog grooming and progress monitoring

JIRA was used as our main organization and work-tracking tool.

Tasks were organized into epics, each representing a high-level feature or objective. These epics were then subdivided into user stories, which captured specific functionality and user requirements.



The team followed agile project management practices, working incrementally through user stories managed in time-boxed sprints. The backlog was continuously refined in JIRA, where user stories were reviewed, prioritized, and updated as requirements evolved.

Progress was tracked by assigning story points to each user story and organizing the work into one-week sprints. During each sprint, task status was monitored directly in JIRA to ensure alignment with planned timelines and sprint goals. Sprint progress was reviewed frequently to confirm that work was advancing as expected.

Story points were estimated using the Fibonacci sequence as inspiration (1, 2, 3, 5, 8, …), as it encourages relative estimation rather than precise measurement. The increasing gaps between numbers reflect the growing uncertainty and complexity of larger tasks, helping the team more accurately assess effort and risk. Story point values were determined through group discussions, allowing the team to reach a shared understanding of each user story's scope and perceived difficulty.

The team held weekly meetings, in addition to in-class working sessions, during which:

- The previous sprint was reviewed, including completed and unfinished user stories
- Any blockers, challenges, or scope changes were identified and discussed
- Upcoming sprints were planned in advance, with story prioritization and point estimates updated in JIRA

This structured routine, alongside tools like XRay, enabled the team to continuously monitor progress, adapt quickly to changes, and maintain a consistent and reliable delivery cadence throughout the project.

This graph presents the burndown chart for **Sprint 3,** which ran from **November 25 to December 2**. It allows us to study the rate at which the story points (and consequently, the associated tasks) were being completed throughout the weeks/sprints.

A recurring pattern observed in our burndown charts is the increased concentration of completed work toward the end of the sprint. This trend can be attributed to factors such as scheduling constraints (including weekends) and the natural progression of task completion as work moves closer to finalization. As a result, a steeper decline in remaining story points is visible in the final parts of the sprint.

Besides that, after looking back and analysing our velocity report, we noticed that in some weeks (particularly the earlier ones) the team attempted to take on more work than it could realistically handle. Over time, however, we started understanding better what the optimal amount of work that should be assigned for each week was, even though some weeks were inevitably busier than others.

In case it is of interest, here is the link to 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.

Regarding code reviewing, Copilot can be a great tool whenever the amount of lines of code is quite large, either to give reviews on what can be improved or to summarize what has been done. However, the changes need to be reflected upon by the author of the pull request, and no pull request shall be accepted without manual review by at least of the other colleagues

2.2 Guidelines for contributors

Coding style

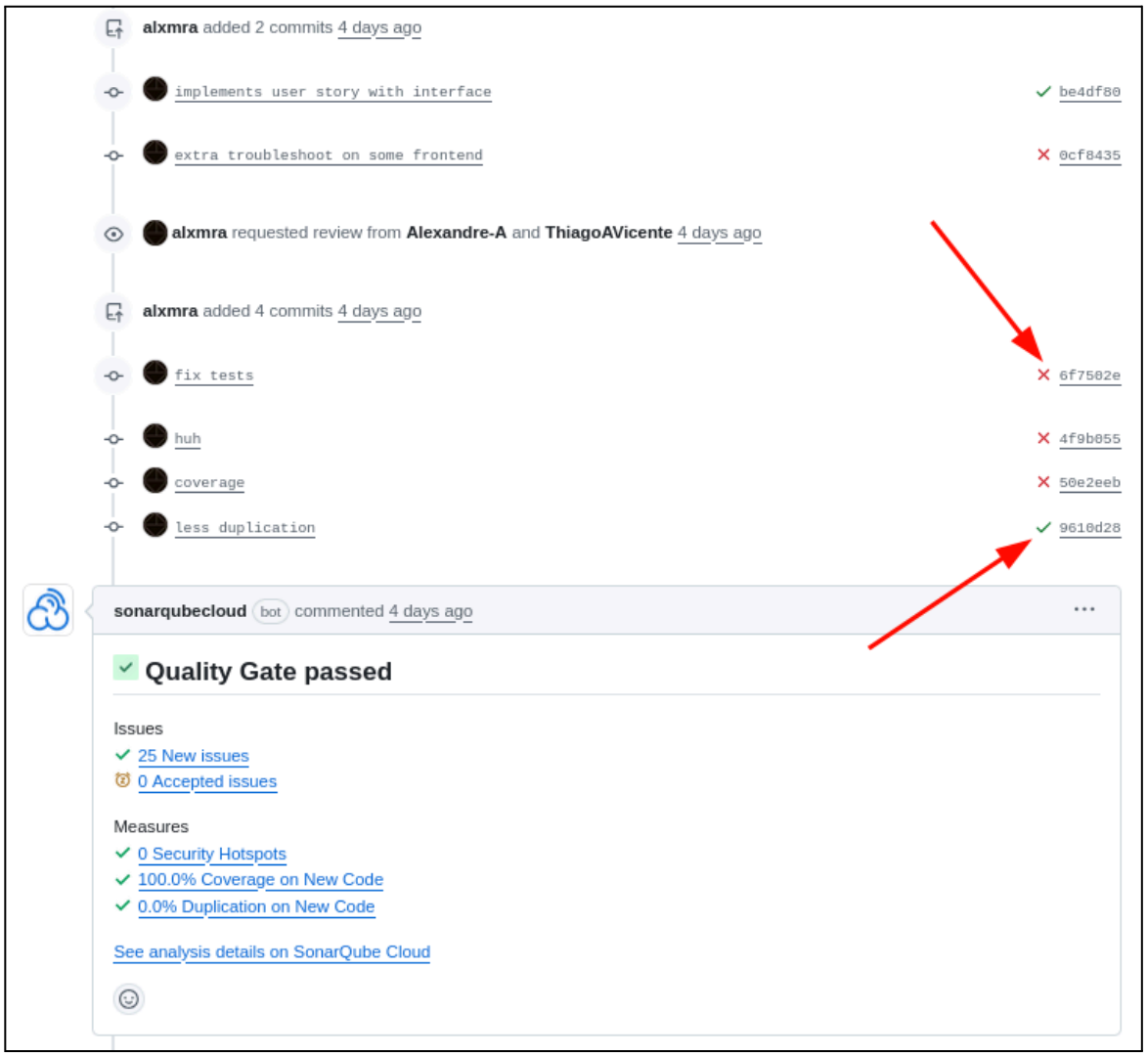
Coding style follows Spring Boot conventions with Java practices, emphasizing consistency above all else. When editing existing code, we first examine the surrounding patterns to match local style preferences, ensuring new code doesn't disrupt the reading rhythm. We use explicit imports rather than wildcard 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.

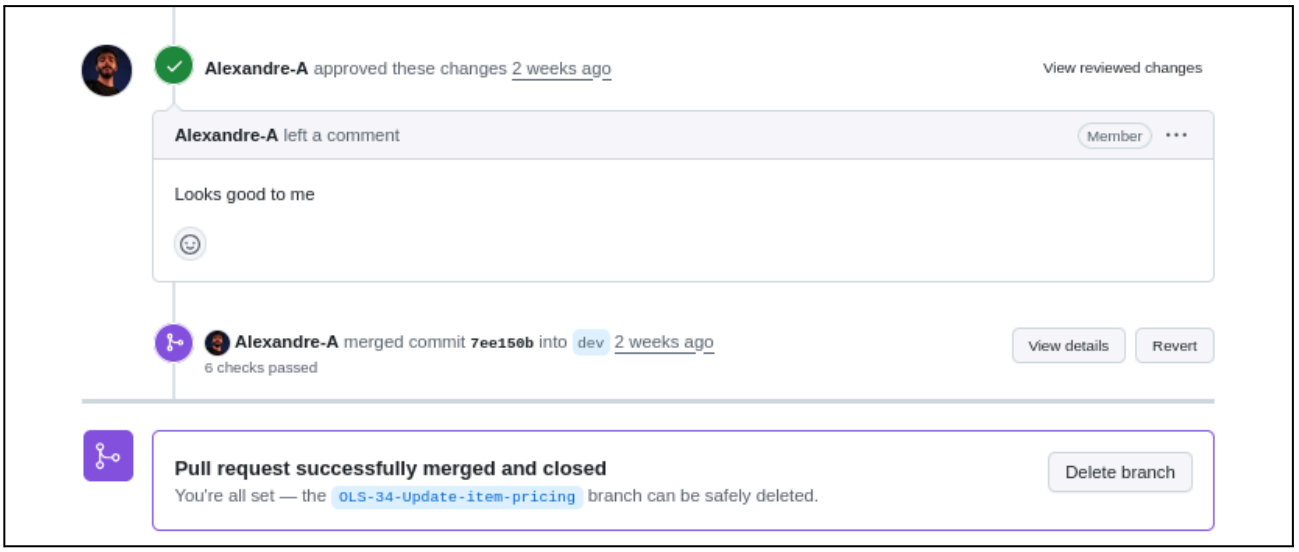
Exception handling is principled, never ignoring exceptions with empty catch blocks without explicit justification and avoiding generic exception catches except in specific top-level or test code scenarios.

Code reviewing

Code reviews are done by the peers. As stated previously, 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.

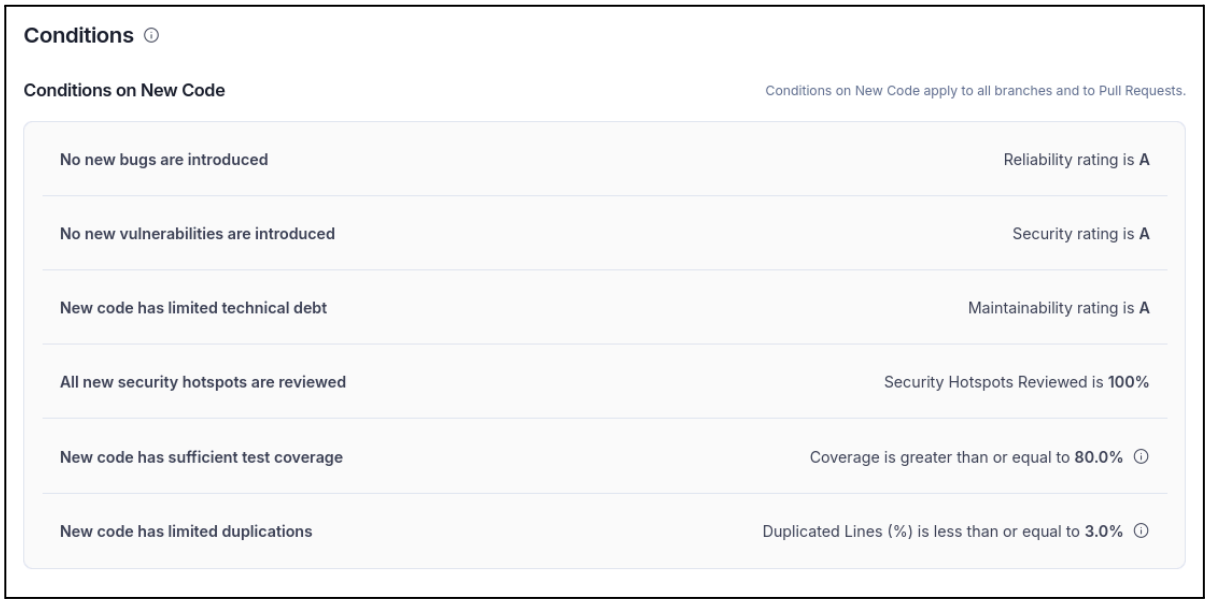
Another developer then reviews the code and can either approve or disapprove. If approved, the pull request may then be merged into the correct branch (usually dev or main).


alxmra added 2 commits 4 days ago
implements user story with interface
be4df80
extra troubleshoot on some frontend
0cf8435
alxmra requested review from Alexandre-A and ThiagoAVicente 4 days ago
alxmra added 4 commits 4 days ago
fix tests
6f7502e
huh
4f9b055
coverage
50e2eeb
less duplication
9610d28
sonarqubecloud bot commented 4 days ago
Quality Gate passed
Issues
25 New issues
0 Accepted issues
Measures
0 Security Hotspots
100.0% Coverage on New Code
0.0% Duplication on New Code
See analysis details on SonarQube Cloud


Alexandre-A approved these changes 2 weeks ago
View reviewed changes
Alexandre-A left a comment
Member
Looks good to me
Alexandre-A merged commit 7ee150b into dev 2 weeks ago
6 checks passed
View details
Revert
Pull request successfully merged and closed
You're all set — the OLS-34-Update-item-pricing branch can be safely deleted.
Delete branch

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:



For the SonarQube Dashboard: SonarQube Dashboard

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

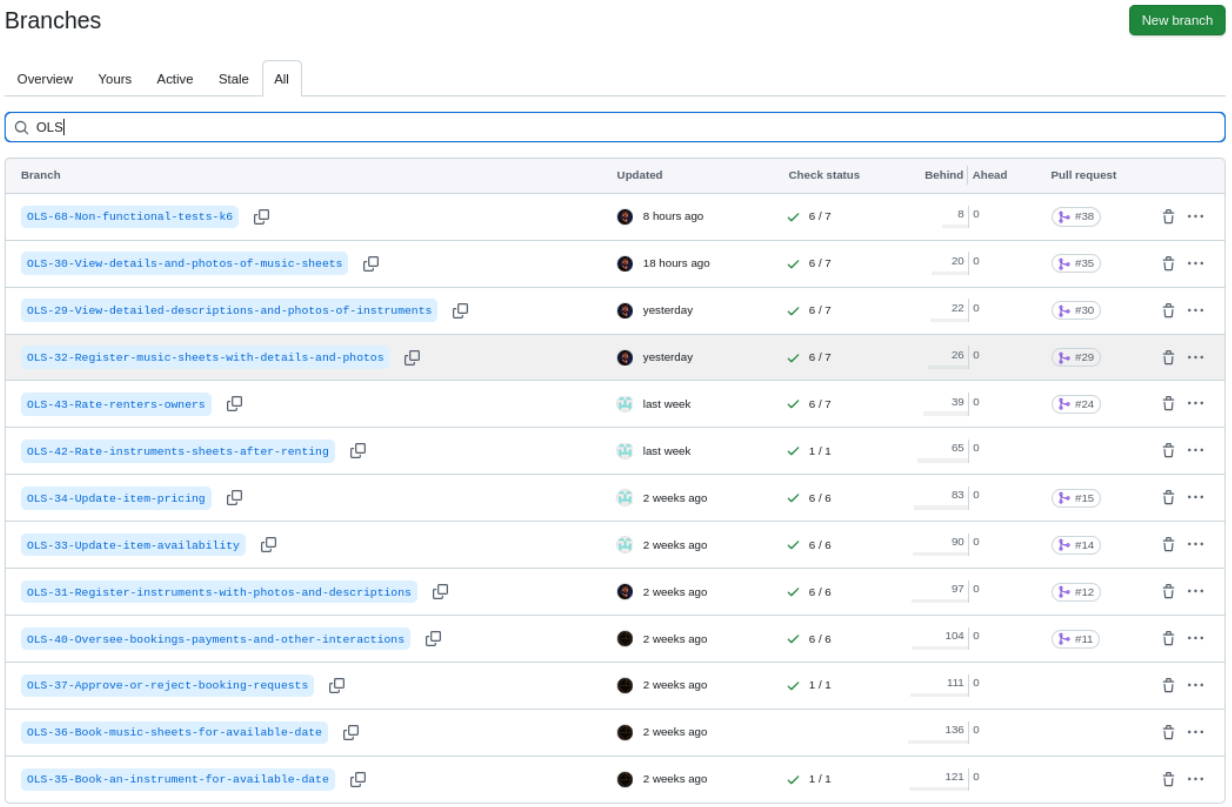
Coding workflow

First of all, tasks/user stories were chosen for each next sprint, then each task/user story was assigned to one developer.

We'll 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. Every feature branch would have the name of the corresponding user story, and branches destined for small fixes would be prefixed by either:
`fix/[name of fix]` or `hotfix/[name of fix]`

When a commit is pushed onto GitHub, SonarQube code analysis is triggered and the results are displayed on the dashboard.



Definition of done

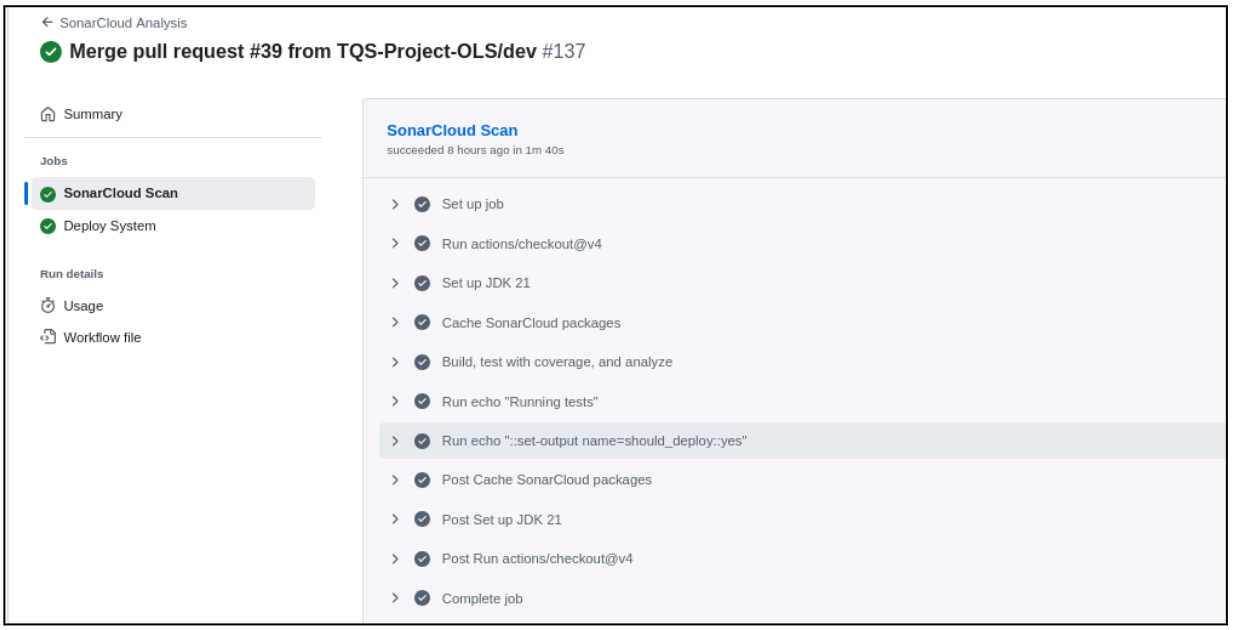
A user story is considered done, when its features are done and it passes all the acceptance criteria defined for it, passes the quality gate and is merged into the dev branch. It is expected to have tests mapping to each acceptance criteria, thus it should pass all those tests.

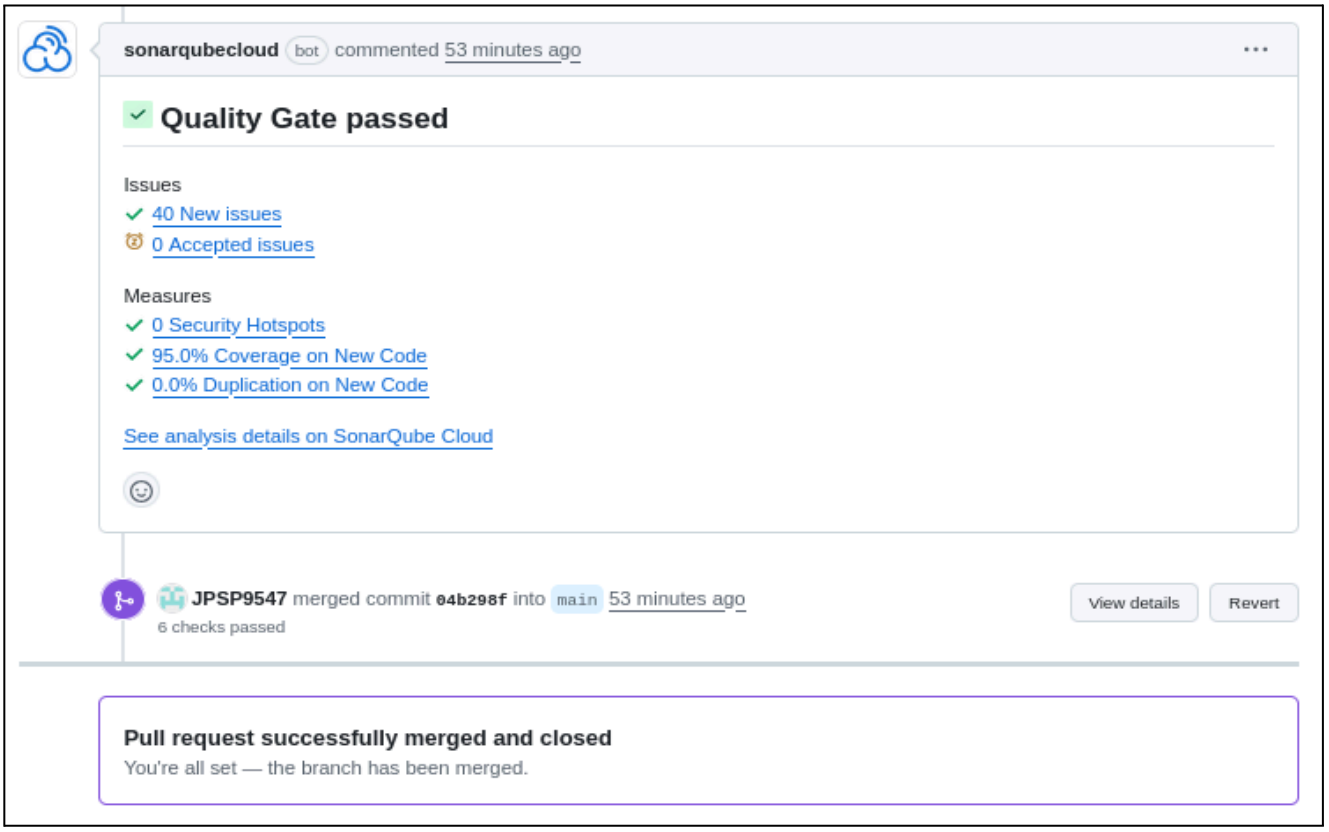
3.2 CI/CD pipeline and tools

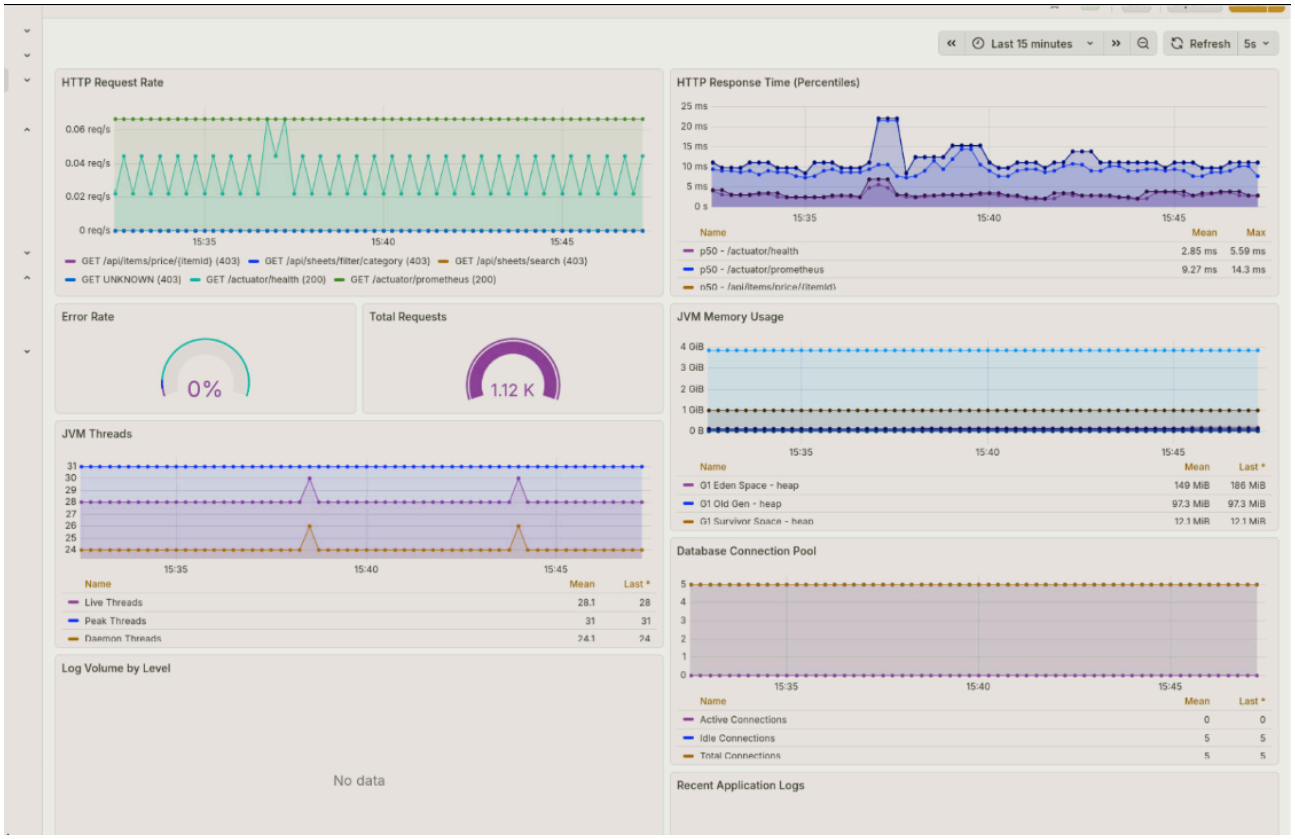
For continuous integration, we have github actions running tests and SonarQube analysis on each new code that is committed, that way we can address bugs and errors early, before they cause damage. Each commit is also automatically analysed by SonarQube to check for code quality.

The analysis also runs 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.

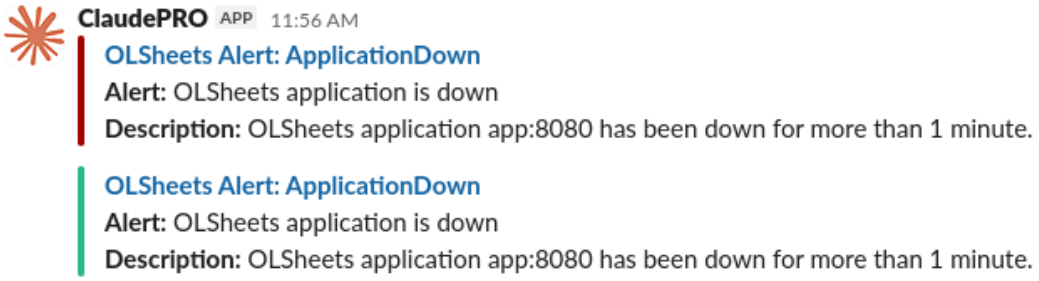

SonarCloud Analysis
Merge pull request #39 from TQS-Project-OLS/dev #137
Summary
Jobs
SonarCloud Scan
Deploy System
Run details
Usage
Workflow file
SonarCloud Scan
succeeded 8 hours ago in 1m 40s
Set up job
Run actions/checkout@v4
Set up JDK 21
Cache SonarCloud packages
Build, test with coverage, and analyze
Run echo "Running tests"
Run echo "::set-output name=should_deploy::yes"
Post Cache SonarCloud packages
Post Set up JDK 21
Post Run actions/checkout@v4
Complete job


sonarqubecloud bot commented 53 minutes ago
Quality Gate passed
Issues
40 New issues
0 Accepted issues
Measures
0 Security Hotspots
95.0% Coverage on New Code
0.0% Duplication on New Code
See analysis details on SonarQube Cloud
JPSP9547 merged commit 04b298f into main 53 minutes ago
6 checks passed
View details
Revert
Pull request successfully merged and closed
You're all set — the branch has been merged.



1

Along with grafana dashboards, an automation for application state was implemented as a Slack bot/App for staying up to date with system status:



The incoherence on the text is due to internal issues on the automation, given that the vertical bar correctly displays the current status of the application, granting developers (or even users, as it is common for most services nowadays to have a ‘status.’ subdomain for availability of status at any time, for anyone, along with sharing on different social media) a 24/7 alerting system well compatible with mobile communication methods.

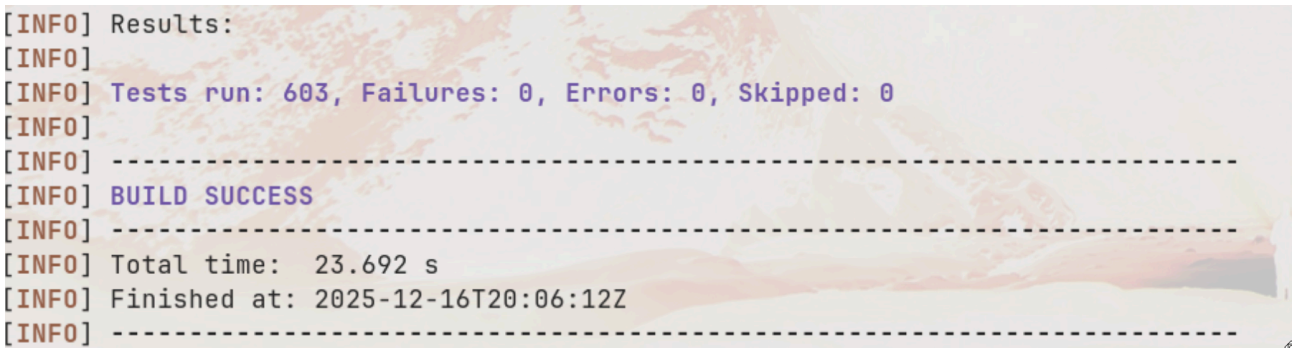
[1] **NOTE:** Colors inverted to light mode for the sake of simplicity of replacement of footage in the report to match the quality standards of keeping the same color scheme constantly.

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.

Source: backend/src/test/java/com/example/OLSHEETS/steps/SearchInstrumentsSteps.java

(...)

```
    @When("I search for instruments with name {string}")
     public void iSearchForInstrumentsWithName(String name) {
         driver.get(FRONTEND_URL);

         // Make sure we're on instruments tab (default)
         WebElement searchInput = wait

.until(ExpectedConditions.visibilityOfElementLocated(By.id("instrument-search
-input")));
         searchInput.clear();
         searchInput.sendKeys(name);

         // Click search button
         driver.findElement(By.id("search-instruments-btn")).click();

         // Wait for results to load - increased timeout
         try {
                    Thread.sleep(3000); // Give more time for async call and
rendering
         } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
         }
     }

     @Then("I should receive {int} instrument(s)")
     public void iShouldReceiveInstruments(int count) {

wait.until(ExpectedConditions.presenceOfElementLocated(By.id("instruments-gri
d")));

         // Wait a bit more for results to render
         try {
             Thread.sleep(500);
         } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
         }

                                              List<WebElement>      results      =
driver.findElements(By.cssSelector(".instrument-result"));
               assertEquals(count,  results.size(),  "Expected  "  +  count  +  "
instrument(s) but found " + results.size());
     }
```

(...)

```
Source: backend/src/test/resources/features/search_instruments.feature

Feature: Search for instruments by name
  As an instrument renter
  I want to search for instruments by name
  So that I can quickly find what I want to rent

(...)

Scenario: Search instruments by partial name match
    Given the following instruments exist:
      | name                 | type          | family    | age | price  |
description           |
      | Yamaha P-125         | Digital Piano | Keyboard  | 2   | 599.99 |
Excellent condition   |
      | Yamaha YAS-280       | Alto Sax      | Woodwind  | 1   | 1299.99|
Professional quality |
      | Fender Stratocaster | Electric      | Guitar    | 5   | 899.99 |
Classic rock guitar   |
    When I search for instruments with name "Yamaha"
    Then I should receive 2 instruments

  Scenario: Search instruments with no matches
    Given the following instruments exist:
      | name                 | type          | family    | age | price  |
description           |
      | Yamaha P-125         | Digital Piano | Keyboard  | 2   | 599.99 |
Excellent condition   |
    When I search for instruments with name "Gibson"
    Then I should receive 0 instruments

(...)
```

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.

Source: backend/src/test/java/com/example/OLSHEETS/unit/BookingServiceTest.java

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

  @Mock
  private BookingRepository bookingRepository;

  @Mock
  private ItemRepository itemRepository;

  @Mock
  private com.example.OLSHEETS.repository.UserRepository userRepository;

  @Mock
        private     com.example.OLSHEETS.repository.AvailabilityRepository
availabilityRepository;

  @InjectMocks
  private BookingService bookingService;

  @Test
  void whenApproveBookingByOwner_thenStatusIsApproved() {
                                          when(bookingRepository.findById(1
L)).thenReturn(Optional.of(booking));
                           when(bookingRepository.findOverlapping(anyLong(),
any(LocalDate.class), any(LocalDate.class)))
      .thenReturn(java.util.Collections.emptyList());
    when(bookingRepository.save(any(Booking.class))).thenReturn(booking);

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

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

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

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

    verify(bookingRepository, times(1)).findById(1 L);
    verify(bookingRepository, never()).save(any());
  }

(...)
```

For integration tests, closed-box developer perspective is the method of approach, but for some Spring Boot integration tests, they might be open-box (e.g. testing how the repository interacts with the DB). These 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 was conducted on the frontend components of the system to ensure the application works as intended from an end-user perspective. Unlike scripted testing, this approach involves unscripted investigation where testers (in this case, the group) freely interact with the application to discover defects that might not be captured through automated test suites. Testers will focus on critical user journeys including the complete rental flow from item discovery through payment, as well as edge cases and error scenarios that may not have been explicitly documented in requirements. The goal is to identify usability issues, visual inconsistencies, and unexpected system behaviors that could impact user experience.

All user stories will be tested in a real production environment to validate functionality under actual operating conditions. This includes testing responsive design across different devices and screen sizes to ensure consistent experience. Testers will pay special attention to the booking process, payment flow, and search functionality - the core paths that users frequently traverse. Issues discovered during exploratory testing were discussed with reproduction steps among developers, including browser information, user actions taken, and expected versus actual behavior, before being taken for resolution. This approach complements the automated testing strategy by providing human insight into usability and real-world usage patterns that scripted tests might miss.

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.

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:

```
time="2025-12-16T10:12:43Z" level=info msg="\n========== Test Summary ==========" source=console
time="2025-12-16T10:12:43Z" level=info msg="Test Type: LOAD" source=console
time="2025-12-16T10:12:43Z" level=info msg="Total Requests: 11072" source=console
time="2025-12-16T10:12:43Z" level=info msg="Avg Response Time: 10.95ms" source=console
time="2025-12-16T10:12:43Z" level=info msg="P95 Response Time: 91.22ms" source=console
time="2025-12-16T10:12:43Z" level=info msg="P99 Response Time: 0.00ms" source=console
time="2025-12-16T10:12:43Z" level=info msg="Error Rate: 0.00%" source=console
time="2025-12-16T10:12:43Z" level=info msg="Failed Requests Rate: 22.22%" source=console
time="2025-12-16T10:12:43Z" level=info msg="Max VUs: 20" source=console
time="2025-12-16T10:12:43Z" level=info msg="==================================\n" source=console

running (8m06.0s), 00/20 VUs, 615 complete and 0 interrupted iterations
default ✓ [ 100% ] 00/20 VUs  8m0s

  scenarios: (100.00%) 1 scenario, 20 max VUs, load
    default: 0m0s duration, up to NaN VUs

  http_reqs.....................: 11072
  http_req_duration.............: avg=10.95ms p(95)=91.22ms
  http_req_failed...............: 22.22%
  successful_requests...........: 11070
  errors........................: 0.00%

running (8m06.0s), 00/20 VUs, 615 complete and 0 interrupted iterations
default ✓ [ 100% ] 00/20 VUs  8m0s
```

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 complete 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%