TQS: Quality Assurance manual

Gonçalo Fonseca [118920], Solomiia Koba [118313], Tiago Coelho [118745], Vasco Pereira [84870]

V2025-12-16

Contents

1 Project management

1.1 Assigned roles

Team Leader
Gonçalo Fonseca

Product Owner
Vasco Pereira

QA Engineer
Tiago Coelho

DevOps Master
Solomiia Koba

Note: Although specific roles were assigned to each team member, all members contributed across multiple areas when necessary.

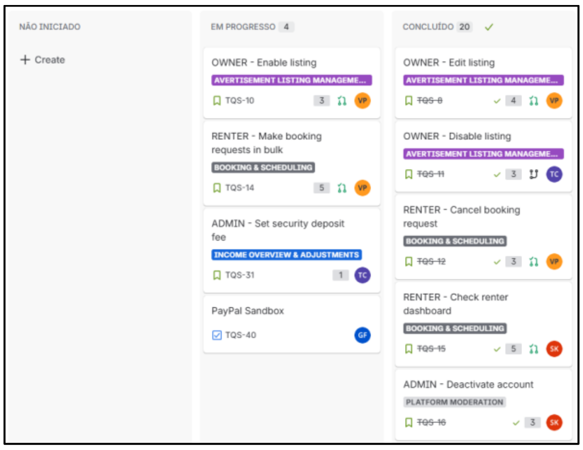
1.2 Backlog grooming and progress monitoring

Work Organization in JIRA

We worked on developing Epics and User Stories. Each Epic represented a major functional area of the system and was broken down into smaller user stories following an agile approach.
Backlog grooming sessions took place mostly during every practical class. During these sessions we reviewed and prioritized user stories, segregate large user stories into smaller ones to follow SMART principles.
Each user story was assigned to a sprint and moved through the default workflow (*To Do → In Progress → In Review → Done*).
Each user story was assigned to a sprint and moved through a defined workflow (e.g., *To Do → In Progress → In Review → Done*). Although roles were assigned, team members frequently collaborated and took ownership of tasks beyond their primary role when required.

Progress Tracking

Progress was tracked regularly using **story points** and **sprint boards** in JIRA. At the beginning of each sprint, the team committed to a set of user stories with an estimated total number of story points.



NÃO INICIADO
+ Create

EM PROGRESSO 4

OWNER - Enable listing
AVERTISEMENT LISTING MANAGEME...
TQS-10 3 VP

RENTER - Make booking requests in bulk
BOOKING & SCHEDULING
TQS-14 5 VP

ADMIN - Set security deposit fee
INCOME OVERVIEW & ADJUSTMENTS
TQS-31 1 TC

PayPal Sandbox
TQS-40 GF

CONCLUÍDO 20 ✓

OWNER - Edit listing
AVERTISEMENT LISTING MANAGEME...
~~TQS-8~~ ✓ 4 VP

OWNER - Disable listing
AVERTISEMENT LISTING MANAGEME...
~~TQS-11~~ ✓ 3 TC

RENTER - Cancel booking request
BOOKING & SCHEDULING
~~TQS-12~~ ✓ 3 VP

RENTER - Check renter dashboard
BOOKING & SCHEDULING
~~TQS-15~~ ✓ 5 SK

ADMIN - Deactivate account
PLATFORM MODERATION
~~TQS-16~~ ✓ 3 SK

Figure 1 – Story Board

Across the **four sprints**, this approach allowed us to measure velocity and improve planning consistency from sprint to sprint.

Requirements-Level Coverage and Testing Monitoring

Proactive monitoring of requirements-level coverage was implemented using Xray integrated with JIRA. User stories were linked directly to test sets, ensuring traceability between requirements and validation activities.

This allowed us to verify that each user story has corresponding test coverage, track test execution status per sprint, and identify untested or partially tested requirements early.

By reviewing Xray reports during and at the end of each sprint, the team ensured that completed user stories met their acceptance criteria and that functional requirements were systematically validated.

2 Code quality management

2.1 Team policy for the use of generative AI

The team adopted a responsible and controlled approach to generative AI tools. We allowed AI tools for code suggestions, refactoring ideas and test case generation whereas it was forbidden to commit AI-generated output without validation, generate tests without verifying their relevance or correctness.

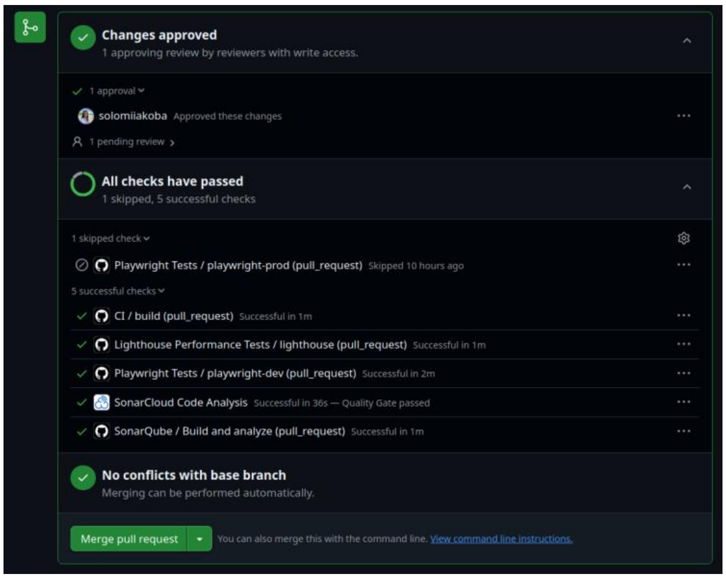
2.2 Guidelines for contributors

Coding style

The project follows standard Java coding conventions, using clear and descriptive naming, small and focused methods, separation of concerns, and consistent formatting enforced by the IDE defaults.

Code reviewing

Code reviews are mandatory and performed via Pull Requests, that are mandatory before merging to dev or main branches. Reviewers should verify tests coverage, correctness of code and compliance with project requirements. The code is approved when all checks and pipelines pass.


Changes approved
1 approving review by reviewers with write access.

✓ 1 approval
solomiiakoba Approved these changes
1 pending review

All checks have passed
1 skipped, 5 successful checks

1 skipped check
Playwright Tests / playwright-prod (pull_request) Skipped 10 hours ago
5 successful checks
✓ CI / build (pull_request) Successful in 1m
✓ Lighthouse Performance Tests / lighthouse (pull_request) Successful in 1m
✓ Playwright Tests / playwright-dev (pull_request) Successful in 2m
✓ SonarCloud Code Analysis Successful in 36s — Quality Gate passed
✓ SonarQube / Build and analyze (pull_request) Successful in 1m

No conflicts with base branch
Merging can be performed automatically.

You can also merge this with the command line. View command line instructions.

Figure 2 – GitHub Action automated checks

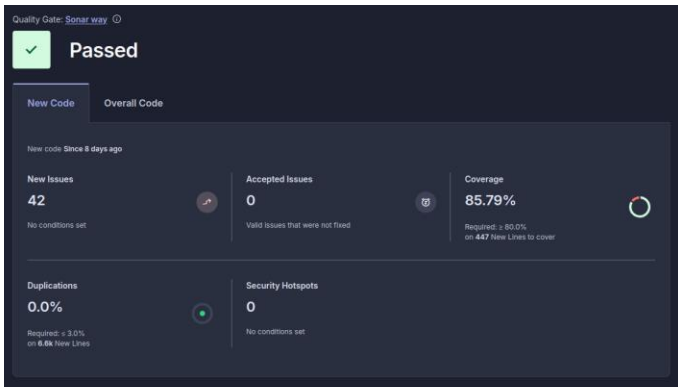
2.3 Code quality metrics and dashboards

Code quality is primarily enforced through successful build, passing automated tests and CI/CD checks. Quality gates include compilation without errors, successful execution of all tests, no failing CI jobs and test coverage. These gates are respectively:

a) SonarCloud: an 80% coverage provides target provides a good safety net

b) Jacoco: is integrated with SonarCloud;

c) OWASP Dependency Check: was also tried to implement but not successfully;


Quality Gate: Sonar way
Passed

New code Since 8 days ago

New Issues
42
No conditions set

Accepted Issues
0
Valid issues that were not fixed

Coverage
85.79%
Required: ≥ 80.0%
on 447 New Lines to cover

Duplications
0.0%
Required: ≤ 3.0%
on 6.6k New Lines

Security Hotspots
0
No conditions set

Figure 3 – SonarQube Report

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow
This project adopts a Continuous Integration and Continuous Delivery (CI/CD) pipeline to ensure code quality, automation, and fast feedback throughout the development lifecycle. The pipeline is

implemented using **GitHub Actions** and integrates build, test, quality analysis, and deployment stages.

Development activities are organized around **user stories managed in Jira**. Each user story represents a unit of work to be completed during a sprint.
During sprint planning, user stories are prioritized, estimated, and assigned to developers. Once a developer is assigned a story, they are responsible for implementing it according to the defined coding and integration workflow.
Traceability between requirements and code is ensured by linking Jira user stories to Git branches and Pull Requests.

Branching Strategy

The team follows a **Gitflow-based workflow** with two main branches:

`main` - Contains stable, production-ready code.

`dev` - Serves as the integration branch where completed features are merged and validated before reaching `main`.

All development work is carried out in **feature branches**, which are created from the `dev` branch.

Working on a user story

Upon being assigned a user story in Jira:

d) A **feature branch is created from dev directly through Jira**.
Jira automatically generates the branch name according to the backlog item, ensuring consistency and traceability.

e) The developer implements the required functionality on this feature branch, committing changes incrementally.

f) Once the implementation is complete, the feature branch is pushed to the remote repository.

Pull Request and Integration

After completing development:

g) A **Pull Request (PR)** is opened targeting the `dev` branch.

h) The PR must:

i) Reference to the corresponding Jira user story.

j) Pass all automated CI checks.

k) Receive **at least one approval** from another team member.

Only after these conditions are met can the Pull Request be merged into `dev`.

Code Review Practices

Code reviews are mandatory for all Pull Requests and serve as a quality assurance mechanism.
Reviewers are responsible for evaluating:

l) Correctness of the implementation

m) Code readability and maintainability

n) Compliance with project standards

o) Test coverage and potential edge cases

Any issues identified during the review process must be addressed before approval is granted.

Definition of done
A user story is considered done when its acceptance criteria are clearly defined and validated using **BDD** practices.
The expected behavior is described in a business-readable form, typically using Cucumber scenarios following the Given–When–Then structure.
All defined scenarios pass, demonstrating that the implemented functionality behaves as expected from the user's perspective.
The implementation is reviewed and approved through the team's Pull Request process.
Once integrated into the shared development branch, the story can be marked as Done.

3.2 CI/CD pipeline and tools

The project implements a **CI/CD pipeline** using **GitHub Actions**, automating build, test, analysis, and deployment processes for every increment.

Continuous Integration (CI)
Every push or Pull Request to dev or `main` triggers the CI pipeline.

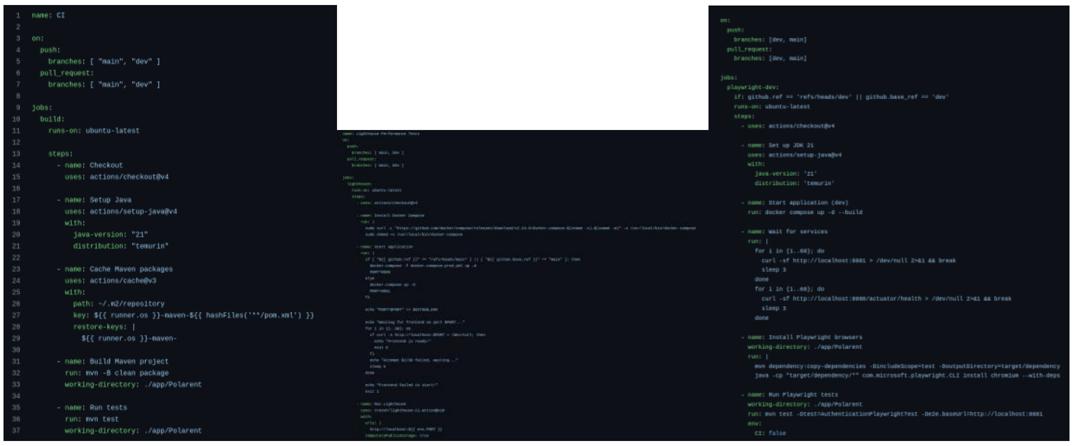
Tools and setup:

p) **Maven** for building the Java project

q) **JUnit** and **Playwright** for unit and end-to-end testing

r) **SonarQube** for static code analysis and quality checks

s) **Lighthouse** for frontend performance validation



```yaml
name: CI

on:
  push:
    branches: [ "main", "dev" ]
  pull_request:
    branches: [ "main", "dev" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          java-version: "21"
          distribution: "temurin"

      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2/repository
          key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
          restore-keys: |
            ${{ runner.os }}-maven-

      - name: Build Maven project
        run: mvn -B clean package
        working-directory: ./app/Polarent

      - name: Run tests
        run: mvn test
        working-directory: ./app/Polarent
```

```yaml
name: Lighthouse Performance Tests
on:
  push:
    branches: [ main, dev ]
  pull_request:
    branches: [ main, dev ]

jobs:
  lighthouse:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install Docker Compose
        run: |
          sudo curl -L "https://github.com/docker/compose/releases/download/[illegible]/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
          sudo chmod +x /usr/local/bin/docker-compose

      - name: Start application
        run: |
          if [ "${{ github.ref }}" == "refs/heads/main" ] || [ "${{ github.base_ref }}" == "main" ]; then
            docker-compose -f docker-compose.prod.yml up -d
            PORT=[illegible]
          else
            docker-compose up -d
            PORT=[illegible]
          fi

          echo "PORT=$PORT" >> $GITHUB_ENV

          echo "Waiting for frontend on port $PORT..."
          for i in {1..30}; do
            if curl -s http://localhost:$PORT > /dev/null; then
              echo "Frontend is ready!"
              exit 0
            fi
            echo "Attempt $i/30 failed, waiting..."
            sleep 5
          done

          echo "Frontend failed to start!"
          exit 1

      - name: Run Lighthouse
        uses: treosh/lighthouse-ci-action@[illegible]
        with:
          urls: |
            http://localhost:${{ env.PORT }}
          temporaryPublicStorage: true
```

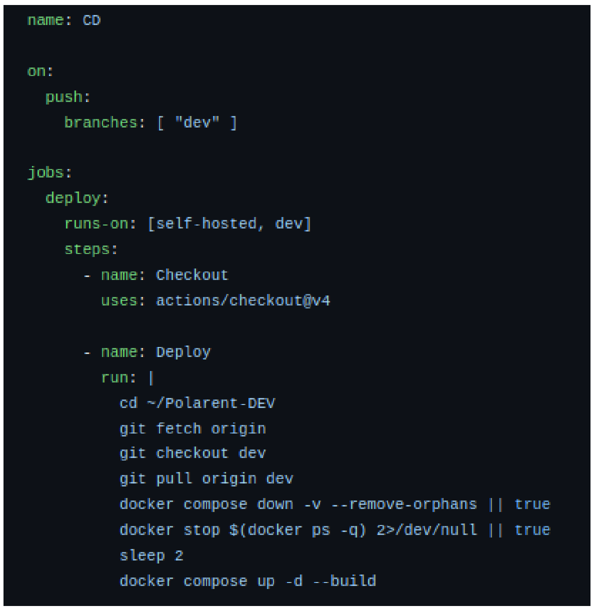
```yaml
on:
  push:
    branches: [dev, main]
  pull_request:
    branches: [dev, main]

jobs:
  playwright-dev:
    if: github.ref == 'refs/heads/dev' || github.base_ref == 'dev'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up JDK 21
        uses: actions/setup-java@v4
        with:
          java-version: '21'
          distribution: 'temurin'

      - name: Start application (dev)
        run: docker compose up -d --build

      - name: Wait for services
        run: |
          for i in {1..60}; do
            curl -sf http://localhost:8081 > /dev/null 2>&1 && break
            sleep 3
          done
          for i in {1..60}; do
            curl -sf http://localhost:8080/actuator/health > /dev/null 2>&1 && break
            sleep 3
          done

      - name: Install Playwright browsers
        working-directory: ./app/Polarent
        run: |
          mvn dependency:copy-dependencies -DincludeScope=test -DoutputDirectory=target/dependency
          java -cp "target/dependency/*" com.microsoft.playwright.CLI install chromium --with-deps

      - name: Run Playwright tests
        working-directory: ./app/Polarent
        run: mvn test -Dtest=AuthenticationPlaywrightTest -De2e.baseUrl=http://localhost:8081
        env:
          CI: false
```

Figure 4 – yml files for CI, Lighthouse and Playwright

The pipeline ensures that all automated tests pass, and that code meets quality standards before merging.

Continuous Deployment (CD)
Changes merged into the dev branch automatically trigger the CD pipeline.
Deployment uses **Docker Compose** to build and run containers in isolated environments.
The pipeline stops existing containers, rebuilds updated images, and deploys the application.

The pipeline ensures that the development environment is always up to date with the latest integrated features.



```yaml
name: CD

on:
  push:
    branches: [ "dev" ]

jobs:
  deploy:
    runs-on: [self-hosted, dev]
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Deploy
        run: |
          cd ~/Polarent-DEV
          git fetch origin
          git checkout dev
          git pull origin dev
          docker compose down -v --remove-orphans || true
          docker stop $(docker ps -q) 2>/dev/null || true
          sleep 2
          docker compose up -d --build
```

Figure 5 – yml file for CD for dev

Environments
The **Development environment** automatically updated on merged to dev (used for integration testing and early feedback)
The **Production environment** code is deployed from the main branch (validated through performance and end-to-end tests before release)

This CI/CD setup allows rapid integration of increments, automated verification of quality, and consistent deployment across environments.

3.3 System observability

Prometheus was used for metrics collection and **Grafana was** used for visualization dashboard of the same metrics. Some of the metrics collected were JVM memory usage and HTTP response time thresholds

4 Continuous testing

4.1 Overall testing strategy

We used layered testing: unit tests with JUnit 5 and Mockito for isolated testing, integration tests with @SpringBootTest and MockMvc with a H2 database, and Cucumber BDD tests with Gherkin feature files for acceptance testing of our user stories.

Testing is integrated into CI/CD through GitHub Actions. Every push and PR triggers mvn test—failing tests block merges. SonarCloud uses JaCoCo to enforce quality gates and track metrics. The CD pipeline only deploys after CI passes, ensuring untested code never reaches deployed environments.
The project follows a **layered testing strategy** to ensure code correctness, integration reliability, and alignment with user expectations.

t) **Unit testing** is performed with **JUnit 5** and **Mockito** to verify individual components in isolation.

u) **Integration testing** uses `@SpringBootTest` and **MockMvc** with an in-memory H2 database to validate interactions between components.

v) **Acceptance testing** leverages **Cucumber BDD** with Gherkin feature files, directly linking tests to Jira user stories and their acceptance criteria.

All tests are **integrated into the CI/CD pipeline** via GitHub Actions:

w) Every push or Pull Request triggers `mvn test`, and failing tests **block merges**.

x) **SonarCloud** tracks coverage with **JaCoCo**, enforcing quality gates.

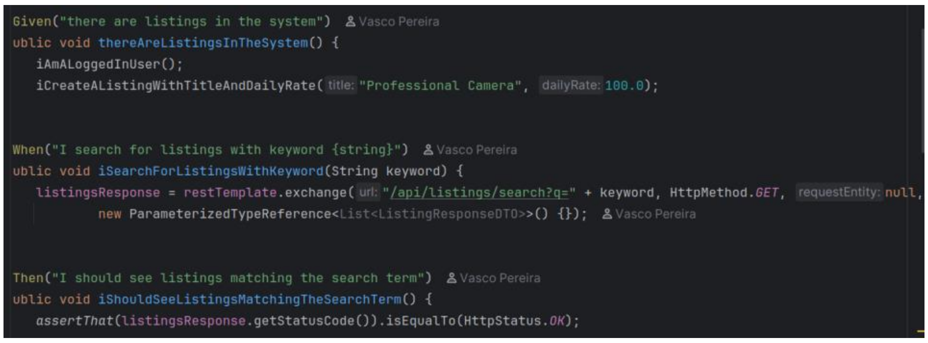
y) The CD pipeline only deploys code that passes all CI checks, ensuring that untested or failing code never reaches deployed environments.

This approach guarantees that testing is continuous, automated, and aligned with both development and deployment processes.

4.2 Acceptance testing and ATDD

Acceptance tests use Cucumber BDD with a closed-box approach. Feature files in Gherkin describe scenarios from the user's perspective, while step definitions use TestRestTemplate to interact with the REST API as an external client—treating the application as a black box.

Tests cover authentication, listings, bookings, and requests. We wrote acceptance tests when implementing new user-facing features or modifying existing workflows, ensuring behavior is validated from the user's perspective.



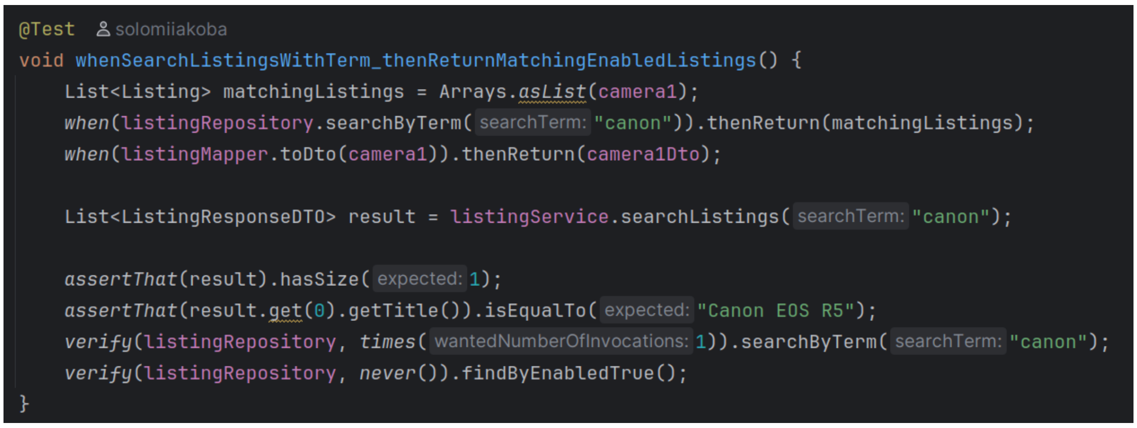
```
Given("there are listings in the system")  Vasco Pereira
ublic void thereAreListingsInTheSystem() {
   iAmALoggedInUser();
   iCreateAListingWithTitleAndDailyRate( title: "Professional Camera",  dailyRate: 100.0);


When("I search for listings with keyword {string}")  Vasco Pereira
ublic void iSearchForListingsWithKeyword(String keyword) {
   listingsResponse = restTemplate.exchange( url: "/api/listings/search?q=" + keyword, HttpMethod.GET,  requestEntity: null,
           new ParameterizedTypeReference<List<ListingResponseDTO>>() {});  Vasco Pereira


Then("I should see listings matching the search term")  Vasco Pereira
ublic void iShouldSeeListingsMatchingTheSearchTerm() {
   assertThat(listingsResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
```

Figure 6 – Example of Given – When - Then

4.3 Developer facing tests (unit, integration)

Unit tests use JUnit 5 with Mockito to test services and controllers in isolation, mocking all dependencies. Developers write unit tests when implementing business logic or modifying existing functionality. Key tests include BookingServiceTest (booking lifecycle and status transitions), ListingServiceTest (CRUD and filtering), and controller tests validating HTTP responses and service delegation.

Integration tests use @SpringBootTest with MockMvc to test the full API request/response cycle against an H2 database. Tests like ListingControllerIntegrationTest and RequestControllerIntegrationTest validate endpoint behavior, JSON responses, and error handling, ensuring all layers work together correctly.



```
@Test  solomiiakoba
void whenSearchListingsWithTerm_thenReturnMatchingEnabledListings() {
    List<Listing> matchingListings = Arrays.asList(camera1);
    when(listingRepository.searchByTerm( searchTerm: "canon")).thenReturn(matchingListings);
    when(listingMapper.toDto(camera1)).thenReturn(camera1Dto);

    List<ListingResponseDTO> result = listingService.searchListings( searchTerm: "canon");

    assertThat(result).hasSize( expected: 1);
    assertThat(result.get(0).getTitle()).isEqualTo( expected: "Canon EOS R5");
    verify(listingRepository, times( wantedNumberOfInvocations: 1)).searchByTerm( searchTerm: "canon");
    verify(listingRepository, never()).findByEnabledTrue();
}
```

Figure 7 – Example of a test

4.4 Exploratory testing

The project has no exploratory testing strategy - it relies entirely on automated, scripted tests.

4.5 Non-function and architecture attributes testing

K6 was used for load testing but was not automated into the pipeline:

z) Load: 10 virtual users for 30 seconds
aa) Thresholds:
bb) Error Rate: < 1% (http_req_failed: ['rate<0.01'])
cc) Response Time: 95th percentile < 500ms (http_req_duration: ['p(95)<500'])

Lighthouse CI was

dd) Triggers: Every push/PR to main/dev branches
ee) Metrics: Core Web Vitals, Performance Score
ff) Storage: Results stored in temporary public storage