TQS: Quality Assurance manual

Tiago Alexandre da Silva Mendes [119773], Cátia Maria Morais Lopes [119087], Diogo Coelho Aires de Nascimento [120031], Daniel Andrade Martins [115868]

Contents

TQS: Quality Assurance manual **1**
1 Project management **1**
1.1 Assigned roles 1
1.2 Backlog grooming and progress monitoring 1

2 Code quality management **2**
2.1 Team policy for the use of generative AI 2
2.2 Guidelines for contributors 2
2.3 Code quality metrics and dashboards 3

3 Code quality metrics and dashboards **3**
3.1 Development workflow 3
3.2 CI/CD pipeline and tools 3
3.3 Artifacts repository [Optional] 4

4 Continuous testing **4**
4.1 Overall testing strategy 4
4.2 Acceptance testing and ATDD 4
4.3 Developer facing tests (unit, integration) 4
4.4 Exploratory testing 5
4.5 Non-function and architecture attributes testing 5

1 Project management

1.1 Assigned roles

- **Product Owner – Cátia Lopes**
- **Team Leader – Tiago Mendes**
- **Quality Assurance (QA) Engineer – Daniel Martins**
- **DevOps Master – Diogo Nascimento**

1.2 Backlog grooming and progress monitoring

Work organization is based on **Agile project management practices**, with development driven by **user stories** maintained in **JIRA**.

- The backlog is organized into **epics and user stories** reflecting the functional and non-functional requirements defined in the Product Specification.
- **Backlog grooming sessions** are conducted regularly to:

- Clarify acceptance criteria
- Split oversized stories
- Re-prioritize work based on risk and dependencies

Progress tracking is performed using:

- Sprint boards to track status (To Do / In Progress / In Review / Done)
- Burndown charts to monitor sprint execution

Requirements-level coverage is monitored by linking user stories to automated tests. When applicable, test management tools integrated with JIRA are used to ensure that critical requirements are validated by acceptance or integration tests.

2 Code quality management

2.1 Team policy for the use of generative AI

The team allows the use of **AI-assisted tools** as **productivity aids**, under strict guidelines.

Allowed uses:

- Suggesting test cases or edge scenarios
- Explaining unfamiliar APIs or frameworks

Not allowed:

- Blindly copying AI-generated code into production
- Using AI-generated content without understanding or reviewing it
- Bypassing security, validation, or testing responsibilities

Policy principle:

AI can assist development, but **developers remain fully responsible** for correctness, security, and quality.

2.2 Guidelines for contributors

Coding Style

The project adopts consistent coding conventions to improve readability and maintainability:

- Clear and descriptive naming for variables, methods, and classes
- Small, focused methods with a single responsibility
- Explicit error handling, especially in booking and payment flows
- Clear separation between business logic, API layers, and persistence

Language-specific conventions follow widely accepted community standards (e.g., Java/Spring or equivalent backend guidelines). Formatting is enforced automatically using linters or formatter tools integrated into the CI pipeline.

Code Reviewing

All changes must be submitted via **Pull Requests (PRs)**.

Code reviews aim to:

- Verify correctness and business logic
- Ensure compliance with coding standards
- Validate test coverage and edge cases
- Detect security or performance issues early

Reviews are mandatory before merging to main branches. AI tools may assist reviewers, but final approval remains a human responsibility.

2.3 Code quality metrics and dashboards

Static code analysis is integrated into the development workflow to continuously assess code quality.
Key practices include:

- Automated analysis during CI builds
- Detection of code smells, bugs, and security vulnerabilities
- Measurement of test coverage

Quality gates are enforced to prevent regressions:

- Builds fail if critical issues are detected
- Minimum test coverage thresholds are required
- New code must not introduce high-severity vulnerabilities

These gates ensure that quality standards remain consistent as the system evolves.

3 Continuous Delivery Pipeline (CI/CD)

3.1 Development workflow

The project follows a **branch-based workflow** aligned with Agile development.

- Developers select a user story from the sprint backlog
- A feature branch is created for each story/couple of stories
- Work progresses with frequent commits
- Automated tests are run locally before pushing changes
- A Pull Request triggers CI validation and review

Definition of Done
A user story is considered done when:

- All acceptance criteria are implemented
- Automated tests pass successfully
- Code review is completed and approved
- CI pipeline completes without errors
- The feature is integrated into the main branch

3.2 CI/CD pipeline and tools

The CI/CD pipeline automates integration and delivery of software increments.
Continuous Integration

- Triggered on Pull Requests and merges
- Includes:
    - Build and dependency resolution
    - Execution of unit and integration tests
    - Static code analysis and quality gate evaluation

Continuous Delivery

- Artifacts are packaged (e.g., container images)
- Deployments are automated to target environments

Environments

- **Development:** rapid feedback, frequent deployments
- **Staging:** production-like environment for validation
- **Production:** stable releases only

Containerization ensures consistency across environments and supports scalability.

3.3 System Observability

To ensure operational reliability, observability is built into the system.

- **Centralized logging** for backend services
- **Health checks** for critical services
- Monitoring of:
  - Error rates
  - Response times
  - Booking and payment failures

Alerts are configured for anomalous conditions, enabling proactive response to system issues.

3.4 Artifacts repository [Optional]

We did not use Maven Artifacts.

4 Continuous testing

4.1 Overall testing strategy

Testing is aligned with the CI/CD pipeline and follows a **continuous testing** approach.
The strategy combines:

- Automated testing at multiple levels
- Early validation of requirements
- Regression prevention through CI enforcement

Tests are treated as first-class artifacts and are required for all significant features.

4.2 Acceptance testing and ATDD

Acceptance tests validate the system from a **user-facing perspective**.

- Tests are derived directly from user stories and acceptance criteria
- Written as black-box scenarios
- Focus on business behavior (e.g., booking flow, listing management)

Acceptance tests are developed alongside features and executed automatically in CI to ensure requirements-level coverage.

4.3 Developer facing tests (unit, integration)

Unit Tests

- Written by developers for all non-trivial business logic
- Focus on isolated components
- Mandatory for new features and bug fixes

Integration Tests

- Validate interactions between services, databases, and external systems
- Include API-level testing for REST endpoints
- Critical flows such as booking and payment are prioritized

API testing ensures contract correctness and data consistency across services.

4.4 Exploratory testing

Exploratory testing is performed manually during development and stabilization phases. Focus areas include:

- Usability issues
- Edge cases not covered by automated tests
- Error handling and unexpected user behavior

Findings are documented and, when relevant, converted into automated regression tests.

4.5 Non-function and architecture attributes testing

Non-functional requirements are validated through targeted tests:

- **Performance testing:** response times, concurrent bookings
- **Reliability testing:** failure handling and retries
- **Security testing:** authentication, authorization, and input validation

These tests ensure compliance with the defined non-functional requirements and support long-term scalability and maintainability.