TQS: Quality Assurance manual

Francisco Albergaria [114646], Gonçalo Sousa [108133] Regina Tavares [114129]

v2025-06-05

1 Project management

1.1 Assigned roles

The team assigned specific responsibilities to each member to ensure a good project management and quality delivery:

Team Manager — Francisco Albergaria

Oversees overall project coordination, facilitates sprint planning and retrospectives, ensures that project milestones are met, and supports team communication and decision making.

Product Owner — Francisco Albergaria

Defines product vision, manages the product backlog, prioritizes user stories based on value and feasibility, and communicates with stakeholders to ensure alignment with user needs.

QA Engineer — Regina Tavares

Defines testing strategy and quality standards, designs and maintains test cases (including acceptance criteria), ensures test automation coverage, and monitors test results through the CI/CD pipeline and Xray integration.

DevOps — Gonçalo Sousa

Sets up and maintains the CI/CD pipeline, configures automated testing and deployment processes, manages Docker-based deployments, and ensures system observability and infrastructure readiness.

1.2 Backlog grooming and progress monitoring

The team used JIRA to manage the product backlog and track project progress.

Backlog grooming

- The Product Owner maintained and continuously refined the product backlog.
- Stories were created, prioritized, and linked to relevant epics and functional areas of the system.
- Acceptance criteria were defined upfront and reviewed during backlog grooming sessions.
- The backlog was regularly updated to reflect evolving project priorities and scope adjustments.

Progress monitoring

- Progress was tracked using story points assigned to each user story during sprint planning.
- The team used burndown charts in JIRA to monitor sprint progress and velocity trends.
- Sprint retrospectives were used to review progress and identify areas for improvement.

Requirements-level coverage monitoring

- The team adopted proactive monitoring of test coverage at the requirements level.
- User stories were linked to corresponding test cases in Xray.
- Automated test executions were integrated with the CI/CD pipeline.
- Test results were synchronized with JIRA, enabling the team to track requirements coverage and identify any gaps.

2 Code quality management

2.1 Team policy for the use of generative AI

The team adopted a pragmatic and responsible approach regarding the use of generative AI assistants (such as GitHub Copilot or ChatGPT) during development.

AI-based tools were allowed to be used as coding assistants, provided that all code generated was properly reviewed and fully understood by the developer before being committed.

Do's:

- Use AI tools to generate boilerplate code (e.g., DTOs, simple mappings).
- Use AI tools to help with test code generation, especially for unit tests and mock setups.
- Use AI tools to accelerate documentation tasks (e.g., writing API doc comments, summarizing test results).
- Always review and validate generated code to ensure it meets project quality and security standards.

Don't's:

- Do not blindly copy-paste AI-generated code without understanding it.
- Do not rely on AI tools for complex business logic or security-critical code.
- Do not use AI tools to bypass the testing process or to "auto-fix" critical findings without peer review.

2.2 Guidelines for contributors

Coding style

The project followed standard Java coding conventions for the backend (Spring Boot) and JavaScript/TypeScript conventions for the frontend (React + Vite):

- Consistent naming conventions (camelCase for variables/methods, PascalCase for classes/components).
- Clear and expressive method and variable names.
- Code formatted using Prettier for frontend and IDE auto-formatting for Java backend.
- No unused code or commented-out code committed to the main branches.
- Exception handling implemented consistently across the backend (standard error responses and logging).
- API endpoints designed following REST resource-oriented best practices.

Code reviewing

Code reviews were an essential part of the development workflow:

All feature branches required a Pull Request (PR) into the dev branch.
PRs could only be merged after approval by at least one reviewer.
Code review checklist included:

- Code quality and adherence to style guides.
- Test coverage and correctness (unit and integration tests).
- Security considerations (no hardcoded secrets, proper input validation, etc.).
- Performance considerations when applicable.
- API documentation updates.

2.3 Code quality metrics and dashboards

The project used SonarQube as the main tool for static code analysis and quality gate enforcement.

SonarQube was integrated into the CI/CD pipeline and ran automatically on each pull request and push to main branches.

Key quality gates defined:

- Coverage: Minimum 80% on new code (unit + integration tests).
- Code smells: No critical code smells allowed.
- Bugs: No critical bugs allowed.
- Security vulnerabilities: No high-severity vulnerabilities allowed.
- Duplications: Max 3% allowed on new code.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

In terms of coding workflow, our team follows these steps:

Story assignment and development process:

- Stories are assigned in JIRA during sprint planning
- Each story is linked to a GitHub issue
- Developer creates a feature branch from dev branch following naming convention: feature/JIRA-ID-short-description
- Development follows TDD (Test-Driven Development) approach:
    - Write tests first
    - Implement feature
    - Ensure all tests pass
    - Run SonarQube analysis locally

Git workflow:

We follow a simple GitFlow approach:

- Main branch: production-ready code
- dev branch: integration branch for features

a) Feature branches: created from dev for each story

- Hotfix branches: created from main if needed

Branch naming conventions:

- Feature branches: feature/JIRA-ID-short-description
- Bug fixes: fix/JIRA-ID-short-description
- Hotfixes: hotfix/JIRA-ID-short-description

Pull Request process:

- Create PR from feature branch to dev
- PR must include:
    - Link to JIRA story
    - Description of changes
    - Test coverage report
    - SonarQube analysis results
- Code review required (minimum 1 reviewer)
- All CI checks must pass:
    - Unit tests
    - Integration tests
    - SonarQube analysis
    - Xray test execution

Code review guidelines:

- Review checklist:
  - Code follows project standards
  - Tests are comprehensive
  - Documentation is updated
  - No security vulnerabilities
  - Performance considerations
- Reviewers should provide constructive feedback
- PR can be merged only after approval

Definition of done

A story is considered "Done" when:

Code Quality:
- Code is written and follows project standards
- All tests are written and passing
- Code coverage meets minimum threshold (80%)
- SonarQube analysis shows no critical issues
- Code is reviewed and approved

Testing:
- Unit tests implemented
- Integration tests implemented
- Xray test cases created and executed
- All tests passing in CI pipeline

Documentation:
- Code is properly documented
- API documentation updated (if applicable)
- README updated (if needed)

Integration:
- Code is merged into dev branch
- CI pipeline passes successfully
- No conflicts with other features

Review:
- Code review completed
- All review comments addressed
- PR approved by at least one reviewer

JIRA:
- Story status updated
- All subtasks completed
- Time logged
- Story moved to "Done" status

3.2 CI/CD pipeline and tools

Our project Continuous Integration and Continuous Delivery (CI/CD) pipeline was designed to ensure code quality, automate testing, and streamline the deployment process.

Continuous Integration pipeline (CI)

The CI pipeline is implemented using GitHub Actions, which automatically triggers on every push to the repository and pull request creation. The pipeline consists of several stages:

Build stage

- The pipeline starts by checking out the code and setting up the Java environment
- Maven is used to build the project and resolve dependencies
- This stage ensures that the code compiles successfully and all dependencies are properly resolved

Test stage

- Unit tests are executed using JUnit
- Integration tests are run to verify component interactions
- Test coverage is measured and reported
- Xray test cases are executed and results are synchronized with JIRA

Code Quality stage

- SonarQube analysis is performed to check code quality
- Code coverage thresholds are verified (minimum 80%)
- Code style and best practices are validated
- Security vulnerabilities are scanned

Artifact generation

- Successful builds generate Docker images
- Images are tagged with build numbers and pushed to GitHub Container Registry
- This ensures that every successful build produces a deployable artifact

Continuous Delivery Pipeline (CD)

The CD pipeline extends the CI process to automate the deployment of our application. We use Docker containers for consistent deployment across environments:

Containerization

- The application is containerized using Docker
- A multi-stage Dockerfile is used to optimize image size
- Base images are regularly updated for security patches
- Environment-specific configurations are managed through environment variables

Deployment process

- Development environment: Automatic deployment after successful CI
- Production environment: Manual approval required
- Deployment is performed using Docker Compose
- Health checks ensure successful deployment

Monitoring and rollback

- Deployment status is monitored

- Automatic rollback on failed deployments
- Logs are collected and analyzed
- Performance metrics are tracked

Tools and Technologies

The CI/CD pipeline is built using the following tools:

- **GitHub Actions**: Orchestrates the entire CI/CD process
- **Maven**: Manages project dependencies and build process
- **JUnit**: Executes unit and integration tests
- **SonarQube**: Performs code quality analysis
- **Xray**: Manages test cases and execution
- **Docker**: Containerizes the application
- **Docker Compose**: Manages multi-container deployments
- **GitHub Container Registry**: Stores Docker images

Pipeline Configuration

The pipeline is configured through YAML files in the .github/workflows directory. The main workflow file defines the stages, conditions, and actions for each step. Environment variables and secrets are managed through GitHub's secure storage.

This CI/CD setup ensures that our code is continuously integrated, tested, and ready for deployment, while maintaining high quality standards and security practices.

4 Software testing

4.1 Overall testing strategy

Our project follows a comprehensive testing strategy that combines multiple testing approaches to ensure high-quality software delivery:

Test-Driven Development (TDD)

We follow TDD principles where developers write tests before implementing features. This approach helps in:

- Early bug detection
- Better code design and maintainability
- Clearer requirements understanding
- Higher test coverage

Testing Implementation:

- Unit Tests (60% of test coverage)
- Integration Tests (25% of test coverage)
- End-to-End Tests (15% of test coverage)

Testing Tools and frameworks:

- JUnit 5 for unit and integration testing
- REST Assured for API testing
- Selenium for end-to-end testing
- Mockito for mocking dependencies
- Cucumber for BDD (Behavior-Driven Development)

Test Automation:

- All tests are automated and integrated into our CI/CD pipeline
- Tests run automatically on every pull request and push to main branches
- Test results are tracked and reported in JIRA through Xray integration

While the project was initially developed following strict Test-Driven Development (TDD) practices, with the first two user stories fully implemented using TDD principles, the team had to adjust the testing approach in later stages of the project.

As the delivery deadline approached and project scope expanded, it became necessary to prioritize feature completion. Therefore, for the subsequent user stories, we adopted a more pragmatic testing workflow:

- Features were implemented first.
- Corresponding unit and integration tests were written immediately after, ensuring that overall test coverage and quality objectives were still met.

4.2 Functional testing and ATDD

Our functional testing strategy focuses on validating the system from a user's perspective:

Acceptance Test-Driven Development (ATDD):

- User stories are translated into acceptance criteria before development
- Acceptance tests are written using Cucumber Gherkin syntax
- These tests serve as living documentation

Functional Testing scope:

- User interface testing
- API endpoint testing
- Business logic validation
- User workflow testing

Testing Process:

- Tests are written during sprint planning
- Developers implement features to satisfy acceptance criteria
- QA team reviews and executes tests
- Results are documented in Xray

4.3 Unit tests

Unit testing is a fundamental part of our development process:

Testing Requirements:

- Every new feature must have corresponding unit tests
- Minimum code coverage requirement: 80%
- Testsmust be independent and repeatable
- Mock external dependencies

Key Unit Test areas:

- Business logic in service layer
- Data access layer
- Utility functions
- Input validation
- Error handling

Testing standards:

- Follow AAA pattern (Arrange, Act, Assert)
- Use meaningful test names
- Include positive and negative test cases
- Document complex test scenarios

4.4 System and integration testing

Our integration testing strategy ensures proper component interaction:

API Testing:

- REST Assured for API endpoint testing
- Test all HTTP methods (GET, POST, PUT, DELETE)
- Validate response codes and payloads
- Test error scenarios and edge cases

Integration Test Scope:

- Service-to-service communication
- Database interactions
- External API integrations
- Message broker operations

Testing Environment:

- Dedicated test environment with test data
- Docker containers for consistent testing
- Automated environment setup

4.5 Non-function and architecture attributes testing

We perform comprehensive non-functional testing.

Performance Testing:

- Load testing using JMeter

- Stress testing for system limits
- Response time monitoring
- Resource utilization tracking

Security Testing:

- OWASP security checks
- Authentication and authorization testing
- Data encryption validation
- Input validation testing

Architecture Testing:

- Component interaction validation
- Scalability testing
- Fault tolerance testing
- Recovery testing

Monitoring and Metrics:

- Performance metrics collection
- Error rate monitoring
- Response time tracking
- Resource utilization monitoring

This testing strategy ensures that our application meets both functional and non-functional requirements while maintaining high quality standards. The integration with our CI/CD pipeline ensures that testing is an important part of our development process.