TQS: Quality Assurance manual

Ana Alexandra Antunes [876543], Bruno Bernardes [876543]

v2025-05-22

[This report should be written for new members coming to the project and needing to learn what are the QA practices defined. Provide concise, but informative content, allowing other software engineers to understand and quickly the practices.
Tips on the expected content (marked in colored text) along the document are meant to be removed.
You may use English or Portuguese; do not mix.]

1 Project management

1.1 Assigned roles

Who oversees what?

1.2 Backlog grooming and progress monitoring

What are the practices to organize the work in JIRA?
How is the progress tracked in a regular basis? E.g.: story points, burndown charts,...
Is there a proactive monitoring of requirements-level coverage? (with test management tools integrated in JIRA)

2 Code quality management

2.1 Team policy for the use of generative AI

Clarify the team position on the use of AI-assistants, for production and test code
Give practical advice for newcomers.
Be clear about "do"s and "Don't"s

2.2 Guidelines for contributors

Coding style
[Definition of coding style adopted. You don't need to be exhaustive; rather highlight some key concepts/options and refer a more comprehensive resource for details. → e.g.: AOS project]

Code reviewing
Instructions for effective code reviewing. When to do? Integrate AI tools?...

Feel free to add more section as needed.

2.3 Code quality metrics and dashboards

[Description of practices defined in the project for *static code analysis* and associated resources.]
[Which quality gates were defined? What was the rationale?]

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow
[Explain, for a newcomer, what is the team coding workflow: how does a developer get a story to work on? Etc...
Clarify the workflow adopted [e.g.. gitflow workflow, github flow . How do they map to the user stories?]
[Description of the practices defined in the project for *code review* and associated resources.]

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
- 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

[What is your team "Definition of done" for a user story?]

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

[Description of the practices defined in the project for the continuous integration of increments and associated resources. Provide details on the tools setup and config.]
[Description of practices for continuous delivery, likely to be based on *containers*]

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.

3.3 System observability

What was prepared to ensure proactive monitoring of the system operational conditions? Which events/alarms are triggered? Which data is collected for assessment?...

3.4 Artifacts repository [Optional]

[Description of the practices defined in the project for local management of Maven *artifacts* and associated resources. E.g.: github]

4 Software testing

4.1 Overall testing strategy

[what was the overall test development strategy? E.g.: did you do TDD? Did you choose to use Cucumber and BDD? Did you mix different testing tools, like REST-Assured and Cucumber?...]
[do not write here the contents of the tests, but to explain the policies/practices adopted and generate evidence that the test results are being considered in the CI process.]

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

4.2 Functional testing and ATDD

[Project policy for writing functional tests (closed box, user perspective) and associated resources. when does a developer need to develop these?

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

[Project policy for writing unit tests (open box, developer perspective) and associated resources:
when does a developer need to write unit test?
What are the most relevant unit tests used in the project?]

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

[Project policy for writing integration tests (open or closed box, developer perspective) and associated resources.]
API 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

[Project policy for writing performance tests and associated resources.]

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.