TQS: Quality Assurance manual

Margarida Ribeiro 119876, Gonçalo Almeida 119792, João Barreira 120054, Rodrigo Santos 119198

v2025-12-15

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 2

3 Continuous delivery pipeline (CI/CD) **2**

3.1 Development workflow 2

3.2 CI/CD pipeline and tools 2

3.3 System observability 3

3.4 Artifacts repository [Optional] 3

4 Software testing **3**

4.1 Overall testing strategy 3

4.2 Functional testing and ATDD 3

4.3 Developer facing testes (unit, integration) 3

4.4 Exploratory testing 3

4.5 Non-function and architecture attributes testing 3

1 Project management

1.1 Assigned roles

For this project, all four listed team members are developers who also carry one additional, assigned project management role.

João Barreira - Team Coordinator

- Ensures fair distribution of tasks among team members.
- Ensures that members work according to the project plan.

Rodrigo Santos - Product Owner

- Represents the interests of the stakeholders.

Margarida Ribeiro - QA Engineer

- Promotes quality-assurance practices within the project.
- Implements and applies instruments to measure the quality of deployments.

Gonçalo Almeida is our DevOps master

- Responsible for development and production infrastructure
- Responsible for required infrastructure configurations

1.2 Backlog grooming and progress monitoring

Our team adopts **agile project management** practices centered around **user stories** as the fundamental unit of work. All development activities are managed through JIRA, which serves as the central repository for requirements, tasks, and progress tracking.

User Stories are formulated following the standard template: “As an Admin, I want to suspend or reactivate user accounts when they violate platform rules so that I can maintain platform safety.”. This structure ensures that each requirement clearly identifies the target user, the desired functionality, and the business value it delivers. Each user story is accompanied by detailed acceptance criteria that define the conditions of satisfaction for the feature. These acceptance criteria serve as the foundation for test case development and validation.

We, as a team, also employ story points as the estimation method for user stories. The estimation sessions involve the entire development team to leverage collective knowledge and promote shared understanding of requirements.

To facilitate high-level planning, related user stories are grouped into epics. The project currently maintains three epics that represent major functional areas for each persona. This hierarchical organization enables us to track progress toward broader project goals while maintaining focus on granular, deliverable user stories within each sprint.

Backlog grooming sessions are conducted one to two times per week, depending on the team’s needs and the evolving requirements. During these sessions, the team reviews upcoming items in the backlog, estimates story points and ensures that the acceptance criteria are well-defined and testable. This way, we can always have a pipeline of ready user stories for subsequent sprints.

JIRA workflow

We follow a standardized workflow within JIRA that reflects the lifecycle of each user story. The workflow consists of four distinct states:

- **To Do**: User stories that have been refined and are ready for development but have not yet been started
- **In Progress**: User stories currently being implemented by a developer
- **In review**: User stories for which development is complete and testing is underway
- **Done**: User stories that have satisfied all acceptance criteria and have been integrated into the main codebase

Jira is integrated with the project's GitHub repository, creating seamless traceability between user stories and code changes. This integration enables automatic synchronization of development activities with project tracking artifacts. When the developers create branches, commit code, or submit pull requests that reference JIRA ticket identifiers, the corresponding user stories are automatically updated into development progress. The integration ensures that all code changes are traceable to specific user stories, supporting both progress monitoring and quality assurance activities.

We operate on a one week sprint cycle, which provides rapid feedback loops from our client and frequent opportunities for course correction. Progress is monitored continuously throughout the week through regular team communication and JIRA board updates.
The team has recognized the value of burndown charts as a visual tool for tracking sprint progress and intends to incorporate them into routing practice.

X-Ray in JIRA

Each user story in JIRA is linked to corresponding test cases in X-Ray. This linkage creates a bidirectional relationship that allows the team to verify that all requirements have associated tests and that all tests trace back to specific requirements. Tests are created before development begins, in alignment with **test-driven development principles**, ensuring that acceptance criteria are translated into executable validation early in the development process.

2 Code quality management

2.1 Team policy for the use of generative AI

The team recognizes generative AI assistants as valuable tools that can enhance productivity, support learning and improve code quality when used responsibly.
The fundamental principle governing our group is **understanding must always precede acceptance**. We, as developers, remain fully responsible for all code committed, regardless of its origin.
Developers may use AI assistants to generate production code, subject to the following requirements and restrictions:

- **All AI generated code must be thoroughly reviewed and understood by the developer before being committed.** Developers must be able to explain the purpose and logic behind every line

of code they introduce into the codebase, whether written manually or generated by AI. Blindly accepting AI suggestions without comprehension is strictly prohibited.
- **Service layer code should be avoided when using AI generation**. The service layer contains core business logic and represents the heart of the application. While AI may be consulted for debugging or exploring approaches, developers should write service layer code manually to ensure deep understanding of business requirements.
- **Frontend code may use AI more freely.** Since frontend development is not the primary focus of this project, AI assistants may be leveraged more extensively for user interface implementation. However, the requirement for review and understanding still applies.
- **All code, including AI generated code, must pass SonarQube quality gates.** Regardless of origin, all committed code is subject to the project's static analysis requirements.

Testing is a core learning objective of this course. As such, we adopt a more conservative approach to AI usage in test development to ensure that all team members gain hands-on experience with testing principles and practices.
- **AI should be used sparingly for test code generation.** While AI can be a useful tool for achieving continuous delivery and maintaining comprehensive test coverage, developers should prioritize manual test development to deepen their understanding on this subject.
- **AI may be used for simple tests, but complex tests must be written manually.** Straightforward unit tests with obvious assertions may be generated with AI assistance. However, tests that validate complex business logic, interaction patterns, or integration scenarios should be written manually.

Prohibited Practices:
- **Copying AI generated code without understanding it.** Developers who cannot explain the code they commit violate the learning objectives of this project.
- **Blindly accepting AI suggestions.** All AI output must be critically evaluated. AI models can produce plausible but incorrect code.

Summary of Guidelines

Practice	Guideline
Production Code Generation	**DO** use AI with thorough review and understanding
Service Layer Code	**DON'T** use AI for service layer implementation
Frontend Code	**DO** use AI freely, with review
Code Autocomplete	**DO** use selectively; **DON'T** accept blindly
Debugging Assistance	**DO** consult AI for troubleshooting guidance
Refactoring Suggestions	**DO** evaluate AI proposals critically
Simple Test Generation	**DO** use AI moderately with full understanding
Complex Test Generation	**DON'T** rely on AI; write manually

Practice	Guideline
Understanding AI Output	**DO** ensure comprehension before committing
SonarQube Compliance	**DO** ensure all code passes quality gates

2.2 Guidelines for contributors

Coding style

The PartyShare project adopts the **Google Java Style Guide** as the foundational coding standard for all Java and Spring Boot code. We are expected to familiarize ourselves with the complete guide, available at https://google.github.io/styleguide/javaguide.html .

While the Google Java Style Guide provides comprehensive coverage of formatting, naming and structural conventions, several key principles warrant explicit emphasis:

- **Naming Conventions** - class names should use 'UpperCamelCade', method and variable names should use 'lowerCamelCase', and constants should use 'UPPER_SNAKE_CASE'. Names should be descriptive and convey intent without requiring extensive comments. Avoid abbreviations except for widely understood terms.
- **Code Organization** - classes should follow a logical ordering; group related methods together and order them from public to private; each class should have a single, well defined responsibility.

Code reviewing

Code review is a mandatory quality gate for all code changes. No code may be merged to the develop branch without approval from at least one team member. The code review process serves multiple purposes: identifying defects, ensuring adherence to coding standards, sharing knowledge across the team and maintaining consistency across the project.

All pull requests must be reviewed and approved by one team member before merging. The reviewer must be someone other than the author developers, who cannot approve their own pull requests. Given that the team members hold multiple roles, the approval should come from someone acting in the **QA Engineer** capacity when possible, as quality assurance is their primary responsibility. However, any team member with appropriate technical expertise may serve as a reviewer.

Reviews must be completed within the same spring in which the pull request is submitted. This ensures that work progresses smoothly and that context remains fresh for both author and reviewer.

What to review

Reviewers should evaluate pull requests across multiple dimensions:

- **Tests** - Verify that appropriate tests have been written and that they adequately cover the new or modified functionality. Check that tests are meaningful, not merely achieving coverage metrics. Ensure that edge cases and error conditions are tested.
- **Business Logic** - Confirm that the implementation correctly satisfies the requirements specified in the acceptance criteria. Validate that the code handles expected inputs appropriately and degrades gracefully for unexpected inputs

- **Quality Gates** - Confirm that SonarQube quality gates have passed. Review any warnings or issues flagged by static analysis and ensure that they have been addressed or appropriately justified. Verify that code coverage metrics meet project thresholds.

Use of AI tools in Code Review

Reviewers are encouraged to use AI tools as assistants during the code review process. **GitHub Copilot** and **SonarQube** can provide valuable insights that augment human judgment. SonarQube automatically analyzes all pull requests and flags potential issues related to code quality, security vulnerabilities and maintainability concerns. GitHub Copilot can assist reviewers in understanding unfamiliar code patterns, suggesting alternative implementations or identifying potential edge cases.

AI tools should enhance, not replace, thoughtful human review. Reviewers must apply their own expertise, contextual knowledge, and professional judgment to evaluate code quality and correctness.

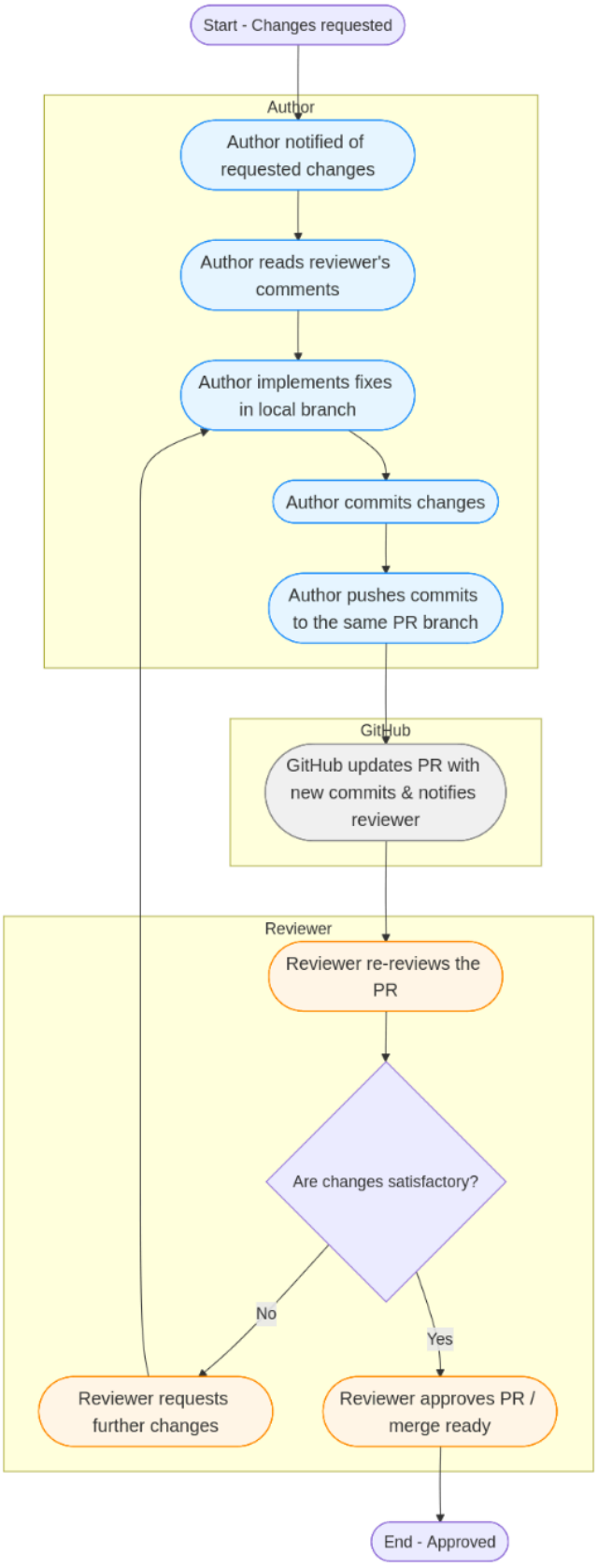
Review Process and Workflow

Code reviews are conducted **asynchronously**. When a pull request is submitted, the author notifies potential reviewers. Reviewers examine the code and provide feedback through GitHub's review interface.

GitHub provides three types of review outcomes:

- **Comment** - Provide feedback without explicit approval or rejection
- **Request Changes** - Indicate that specific issues must be addressed before the PR can be merged
- **Approve** - Confirm that the code meets quality standards and is ready for integration

When changes are requested, the following workflow applies:



Once final approval is granted, the **author** is responsible for merging the pull request. The project uses a **merge commit** strategy, which preserves the complete commit history of the feature branch and creates an explicit merge commit in the develop branch. After merging, the feature branch may be deleted to keep the repository clean.

2.3 Code quality metrics and dashboards

The project employs **SonarCloud** as the primary platform for static code analysis, code quality monitoring, and technical debt management. Static analysis is fully integrated into the project's continuous integration pipeline through GitHub Actions. Analysis is triggered automatically on every push to the **main** and **develop** branches, as well as on all pull requests targeting these branches. This automation ensures that code quality is evaluated consistently and that quality issues are identified early in the development lifecycle.
While continuous integration provides automated quality checks, developers are encouraged to run SonarQube analysis locally during development. Local analysis enables developers to identify and resolve quality issues proactively before pushing code, reducing the iteration cycle during code review.

This project employs the **default SonarCloud** quality gate, which represents industry best practices for code quality thresholds.
The Sonar way quality gate enforces the following conditions on new code:

- **Coverage** - Code coverage on new code must be at least 80%
- **Duplicated Lines** - Duplicated lines in new code must not exceed 3%
- **Maintainability Rating** - New code must achieve a maintainability of A
- **Reliability Rating** - New code must achieve a security rating of A

Code coverage is measured using **Jacoco**, which is integrated into the Maven build process. The project has configured a code coverage target of **80%** in the pom.xml configuration. This threshold ensures that the majority of the codebase is validated through automated tests. The 80% balances thoroughness with practicality, acknowledging that certain code may not warrant exhaustive testing.
Coverage reports are generated during the `verify` phase of the Maven build and are automatically uploaded to SonarCloud for visualization and trend tracking. Developers can also view detailed coverage reports locally by examining the HTML reports generated in the `target/site/jacoco` directory.

Beyond the quality gate thresholds, we also actively monitor several additional metrics to maintain awareness of code health and identify areas of improvement:

- Code Smells
- Bugs
- Duplicated Code

These metrics are accessible through the SonarCloud dashboard and are reviewed as part of the team's quality monitoring activities.

Handling Quality Gate Failures

Quality gate failures represent a mandatory quality checkpoint that must be resolved before code can be merged. Our policy is that **all code must pass the SonarCloud quality gate prior to merge**. This non-negotiable standard ensures that quality does not erode incrementally and that the main/develop branches always represent a known, acceptable level of quality.
When a pull request fails the quality gate, the author must investigate the failing conditions, address the underlying issues, and push updated commits that satisfy the quality requirements. The reviewer is responsible for verifying that the quality gate has passed as part of their review checklist.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

This project follows a structured development workflow that ensures quality, traceability and collaboration at every stage. For developers new to the team, understanding this workflow is essential to contributing effectively and maintaining the project's quality standards.

The development process centers on **user stories** as the fundamental unit of work. Each user story represents a "slice" of functionality with clear acceptance criteria. All development activities are organized around delivering these user stories in short, iterative cycles.

We adopt a **feature branch** workflow based on **GitHub Flow** principles, in order to support continuous integration and structured quality gates. The workflow uses two primary branches:

- **develop** - the main development branch that serves as the integration point for all feature work. This branch represents the current state of ongoing development and is the source for creating feature branches
- **main** - the stable production branch that contains only thoroughly tested, reviewed and approved code.

Branching Model

1. **Feature branches are created from 'develop'** - When a developer begins work on a user story, they create a new feature branch based on the current state of the 'develop' branch
2. **Development occurs in isolation** - All implementation, testing and refinement happens within the feature branch
3. **Pull requests (PRs) merge into 'develop'** - When the feature is complete and ready for integration, the developer creates a PR that merges the feature branch into 'develop'
4. When develop is ready for release, a release is created and put into 'main' **by the DevOps**

User Story to Production Code

For a developer joining the team, the typical workflow for implementing a user story proceeds through the following stages.

At the beginning of each sprint, the **Team Coordinator** facilitates sprint planning. During this session, the team reviews the refined backlog, discusses upcoming user stories and distributes work for the week. The Team Coordinator is the one who assigns stories to developers.

Each developer leaves sprint planning with a clear understanding of which user stories they are responsible for and what the acceptance criteria require.

1. Before beginning development, the developer creates a **feature branch** from the `develop` branch. The branch name must follow the format.
   Format: <prefix>/<ISSUE-KEY>-<description>.

2. With the feature branch created, the developer implements the functionality specified in the user story's acceptance criteria. Development proceeds iteratively, with frequent commits that capture logical increments of progress. Commits should be atomic and logically coherent, representing discrete units of progress. The commit history should tell a clear story of how the feature was developed.

Format: “<ISSUE-KEY> <type>: <message summary>”
The type can be:
- feature
- fix
- refactor
- test
- docs
- chore

3. Before pushing code to the remote repository, the developer runs a comprehensive set of local quality checks.
   Command: ./mvn clean test;
   Local SonarQube analysis should be performed before committing any new code.
   The developer can also examine detailed coverage reports. If local checks reveal issues, failing tests, insufficient coverage or quality gate failures the developer **must** address them before pushing the code.
4. When the feature is complete and all acceptance criteria are satisfied, the developer creates a pull request that proposes merging the feature branch into `develop`.
   Title Format: <ISSUE-KEY>: <Descriptive Title>
   The pull request description provides comprehensive context for reviewers and should include:
   - A link to the JIRA issue
   - A summary of what was implemented
   - A description of key changes
   - Confirmation that tests were written and pass
   - Confirmation that SonarQUbe quality gates pass

   Creating the pull request automatically triggers the GitHub Actions CI pipeline, which runs the full test suite and SonarCloud analysis.

Once the pull request is created, the workflow is the one described on section **2.2 Guidelines for Contributors** under **Code Review.**

Note: Every branch name, commit message, and pull request title **must** include the JIRA Issue Key

Definition of done
We maintain a clear, unambiguous **Definition of Done (DoD)** that specifies when a user story is considered complete and ready for production. All user stories must satisfy every criterion in the DoD before they can be moved to "Done" status in JIRA.
A user story is considered **DONE** when:
- ☐ **All acceptance criteria is met** - The implemented functionality satisfies every condition specified in the user story’s acceptance criteria.
- ☐ **Code is implemented and reviewed** - The feature has been **fully** implemented and the code has undergone thorough review by at least one team member who has approved the pull request.
- ☐ **Unit tests are written and passing** - Comprehensive unit tests have been created that validate the functionality at the component level.
- ☐ **Code coverage meets the 80% threshold**
- ☐ **SonarCloud quality gate has passed**
- ☐ **Code is merged to the develop branch**
- ☐ **X-Ray tests linked and passed** - All Xray Test and Test Execution records linked to the user story in JIRA show passing status

☐ **JIRA issue is moved to "Done" status**

This DoD ensures that "Done" has consistent and objective meaning across all user stories, preventing premature closure of incomplete work.

3.2 CI/CD pipeline and tools

The project employs a continuous integration and continuous delivery (CI/CD) pipeline that automates building, testing, quality analysis and deployment processes.
The CI/CD infrastructure is built on **GitHub Actions**, the pipeline integrates multiple tools including Maven for build management, JUnit and Jacoco for testing and coverage, SonarCloud for static analysis and X-Ray for test management and requirements traceability.

The CI/CD pipeline is defined in the `. github/workflows/ci-cd. yml` file and is triggered by the following events:

- **Push Events** - The pipeline executes automatically when code is pushed to the 'main', 'develop' or any 'feature/**' branch.
- **Pull Request Events** - The pipeline executes when a pull request is opened or updated targeting the 'main' or 'develop' branches.

The intentional triggering on both feature branch pushes and pull requests allows for developers to receive early feedback as they work and for reviewers to confidently assess pull requests knowing automated checks have already validated the changes.

Continuous Delivery

The project also employs Docker and Docker Compose for containerization, enabling consistent deployment across different environments.
The backend application container is built from a custom Dockerfile. It uses an unless-stopped restart policy to ensure automatic recovery from failures and maps external port 80 to internal port 8080. The container includes a health check that monitors the /actuator/health endpoint every 30 seconds with a 10 second timeout.
The database container runs PostgreSQL 15 under the name **tqs-postgres**. It uses a named volume **postgres_data** to ensure data persistence across container restarts.The database health check uses the **pg_isready** command and runs every 10 seconds with a 5 second timeout and 5 retry attempts.
The network configuration employs a bridge driver network named **tqs-network** with a subnet of 10.11.0.0/24. Container orchestration is carefully managed to ensure the backend application waits for the PostgreSQL health check to pass before starting, preventing connection errors during initialization.

Database Migration Strategy

Flyway is configured for production with several important settings. It is enabled by default, with baseline-on-migrate set to true to allow migrations on existing databases starting from baseline version 0. This configuration ensures several critical capabilities for database management. Flyway provides automated schema versioning, enabling zero-downtime deployments through carefully orchestrated schema changes. Data persists across container restarts thanks to the PostgreSQL volume, and the database state remains consistent across all environments through version controlled migration scripts.

Deployment Environments

- **Production Environment**
  - Latest code is checked out from the main branch
  - Environment configuration is established by creating a .env file from GitHub Secrets and populating all required environment variables
  - Existing containers are shut down using **docker-compose down**, which preserves all data in the PostgreSQL volume
  - Fresh images are built and containers are started in detached mode using **docker-compose up -d –build**
  - Unused Docker resources are cleaned up with **docker system prune -f** to maintain system efficiency

Deployment is triggered automatically whenever code is pushed to the main branch, and can also be initiated manually via workflow dispatch.

- **Development Environment**

The development environment is configured through the **docker-compose.dev.yml** file and is designed for local development with hot-reload capabilities. Developers can use either an H2 in-memory database or local PostgreSQL, with deployment occurring manually as developers initiate it.

The Party Share project implements a comprehensive CI/CD pipeline that ensures code quality, test coverage and automated deployment. The multi-stage pipeline validates every change through automated testing, static analysis and quality gates before deployment to production. The containerized deployment strategy using Docker ensures consistency across environments, while Flyway manages database schema evolution safely.

3.3 System observability

The application integrates **Sentry**, a real-time error tracking and monitoring platform that automatically captures exceptions, performance issues and application crashes. To enhance error diagnostics, the project integrates the Sentry Maven Plugin version 0.10.0, which uploads source code context to Sentry. This allows developers to view the exact code lines where errors occurred directly within the Sentry dashboard.

Events and Alarms Triggered

Sentry automatically captures and reports several categories of events without requiring manual instrumentation.

- **Uncaught Exceptions**
- **HTTP Errors**
- **Performance Issues**
- **Crash Reports**

The observability infrastructure collects diverse data to support proactive monitoring and rapid incident response:

- Error Data (Sentry): Exception types and frequencies, stack traces with source code context, user sessions affected by errors, performance transaction traces, database query performance, HTTP request latency distribution, and error trends over time.
- Metrics Data (Actuator): JVM memory and GC metrics, HTTP throughput and latency, database connection pool utilization, thread pool statistics, system resource usage (CPU, disk), and custom business metrics.
- Health Data (Docker): Container health status transitions, health check success/failure rates, application startup duration, database connectivity status, and service availability percentage.

- Log Data (SLF4J/Logback): User authentication audit trail, business transaction records, input validation failures, exception details with context, performance-related warnings, and security events.

The PartyShare project implements a comprehensive observability strategy that provides visibility into application health, performance, and operational behavior. Sentry captures and tracks errors with rich contextual information, Spring Boot Actuator exposes operational metrics and health status used by Docker for automatic recovery, SLF4J logging creates an audit trail of business operations and system events at environment-appropriate detail levels, and Docker health checks ensure container availability with automatic restart capabilities.

4 Continuous testing

4.1 Overall testing strategy

The testing strategy follows a test pyramid approach with three distinct layers, each serving specific purposes and providing different types of confidence:

- **Unit Tests (Base Layer)**: The foundation consists of isolated unit tests that verify individual components in isolation.
- **Integration Tests (Middle Layer)**: Integration tests verify that multiple components work correctly together, particularly focusing on controller-service-repository interactions and database operations.
- **End-to-End Tests (Top Layer)**: E2E tests validate complete user workflows through the actual web interface using Playwright for browser automation. These tests ensure that all system components, including the frontend, backend, database, and business logic, work correctly together from the user's perspective.

Testing Tools and Frameworks

- JUnit 5
- Mockito
- Spring Boot Test
- MockMvc
- Cucumber
- Playwright
- AssertJ
- Maven Surefire Plugin
- Maven Failsafe Plugin
- JaCoCo

Testing-Driven Development Practices

Although the project may not have adhered to a strict, pure "test-first" methodology for every line of code, the testing practices employed demonstrably embrace the **principles and mindset of Test-Driven Development (TDD)**.

Behavior-Driven Development (BDD) with Cucumber

The project extensively employs BDD practices using Cucumber to bridge the gap between technical implementation and business requirements. 10 feature files define system behavior in Gherkin syntax, making requirements executable and self-documenting.

Every CI pipeline execution produces detailed logs showing test execution progress. The console output displays the number of tests run, number of tests passed/failed, execution time per test class, and overall test phase duration. Failed tests include full stack traces with assertion failures, enabling rapid diagnosis without artifact downloads.
X-Ray automatically creates test execution records in Jira for each CI pipeline run. Each execution shows the timestamp of the test run, Git commit SHA, branch name, test pass/fail status per scenario, and links to corresponding GitHub Actions workflow runs. Project managers and stakeholders can view test execution trends over time, identify flaky tests with inconsistent results, and track requirement test coverage without accessing GitHub.
JaCoCo generates comprehensive coverage reports accessible as GitHub artifacts and via SonarCloud dashboards. The reports include overall project coverage percentage, coverage by package (controller, service, dto, entity), coverage by file with visual line highlighting, branch coverage showing conditional logic testing, and coverage trends across commits.

4.2 Acceptance testing and ATDD

This project follows Acceptance Test-Driven Development principles using Behavior-Driven Development (BDD) with Cucumber. Acceptance tests are written from a closed-box, user-facing perspective, validating system behavior without knowledge of internal implementation details. Tests interact with the application exclusively through the user interface using Playwright browser automation, treating the entire backend as a black box.
Developers must write acceptance tests before implementing user stories as part of the ATDD workflow.
The ATDD approach with Cucumber and Playwright ensures that acceptance criteria are defined collaboratively, written before implementation, expressed in stakeholder-friendly language, validated automatically in CI/CD, and traceable to user stories via X-Ray integration.

4.3 Developer facing tests (unit, integration)

Unit tests in the PartyShare project follow an open-box (white-box) testing approach, validating internal implementation details and code paths from a developer's perspective. Tests have full knowledge of the component's internal structure, dependencies, and implementation logic, using this knowledge to design targeted test cases that exercise specific code branches, edge cases, and error handling paths.
Unit tests employ comprehensive mocking to isolate the component under test from its dependencies.

When to write Unit Tests?

Developers write unit tests before feature implementation.The project follows a test-driven mindset where tests are created to verify correctness as development progresses.

Integration tests validate that multiple components work correctly together, using a combination of open and closed-box approaches. Tests have visibility into the application's architecture and use this knowledge to configure test scenarios, but they validate outcomes based on observable system behavior rather than internal state inspection.
Unlike unit tests, integration tests use real Spring beans and actual database persistence (H2 in-memory database) to ensure that components integrate correctly. Only external services (payment gateways, email providers) are mocked to maintain test determinism and avoid external dependencies.

When to write Integration Tests?

- **Complex Workflows** - Features involving multiple service layers and database transactions
- **Database Interactions** - Custom queries, complex joins and repository methods with multiple conditions are validated through integration tests that execute actual SQL against a test database
- **API Contracts** - REST API endpoints receive integration tests to verify that the entire request processing pipeline functions correctly.

API Testing

REST controller tests using **@WebMvcTest** validate API endpoints in isolation with mocked services. These tests verify request deserialization, business logic invocation, response serialization, HTTP status code correctness and error response formatting.
REST controller integration tests using **@SpringBootTest** validate API workflows with real database persistence, transaction management and actual data validation. These tests send JSON payloads, verify database state changes and confirm response accuracy.

The developer-facing test strategy employs unit tests for fast, isolated validation of individual components using MockMvc and Mockito, and integration tests for multi-component workflow validation with real Spring context and database persistence. API tests ensure REST endpoints adhere to contracts and handle errors correctly. Developers write tests during implementation as code is developed, and all tests execute automatically in CI/CD with results flowing to coverage analysis (JaCoCo/SonarCloud) and test management (X-Ray).

4.4 Exploratory testing

Exploratory testing occurs informally during development rather than as scheduled test sessions. Developers and the QA engineer perform manual exploration when:

- **Validating New Features** - After implementing a user story and ensuring automated tests pass, developers manually interact with the feature through the live application to verify user experience, visual presentation, and workflow intuitiveness. This includes testing the feature on the deployed environment at deti-tqs-08.ua.pt to validate behavior in production-like conditions.
- **Reviewing Pull Requests** - Code reviewers may manually test features locally by checking out the feature branch, starting the application with docker-compose up, and exploring the functionality to ensure it meets acceptance criteria and doesn't introduce regressions.

4.5 Non-function and architecture attributes testing

The project implements load testing to validate application performance under realistic and stress conditions. Load tests evaluate critical user workflows to ensure the application can handle expected traffic volumes while maintaining acceptable response times.
Google Lighthouse audits were also conducted to evaluate frontend performance, accessibility, best practices, and SEO metrics.

The PartyShare project implements a comprehensive approach to non-functional testing, combining load testing for backend performance validation, Lighthouse audits for frontend

optimization, and continuous observability through Actuator and Sentry. Security is validated through static analysis (SonarCloud), dependency scanning, and authorization tests. Reliability is ensured via health checks, transaction management, error handling tests, and automated deployment.