TQS: Quality Assurance manual

José Rubem Lins de Aquino Neto [101092]
v2026-01-29

Contents

1 Project management

1.1 Assigned roles

Not Applicable

1.2 Backlog grooming and progress monitoring

Not Applicable

2 Code quality management

2.1 Team policy for the use of generative AI

Policy Statement:The team permits the use of AI assistants (e.g., GitHub Copilot, ChatGPT, Claude) for both production and test code development, subject to the following guidelines:

Do's:

- Use AI to accelerate boilerplate code generation (DTOs, mappers, repetitive tests)
- Use AI to suggest test cases and edge cases you may have overlooked
- Use AI as a learning tool to understand unfamiliar APIs or patterns
- Always review and understand AI-generated code before committing
- Use AI to help refactor and improve code quality
- Run all tests and linting before committing AI-assisted code

Don'ts:

- Never commit AI-generated code without understanding what it does
- Don't rely on AI for security-critical logic without expert review
- Don't use AI to generate secrets, API keys, or credentials
- Don't copy AI output that includes copyrighted or licensed code without attribution
- Don't skip code review for AI-generated contributions

Newcomer Advice: AI tools are excellent for productivity, but they can introduce subtle bugs or outdated patterns. Always validate AI suggestions against project conventions, run the full test suite, and ensure SonarCloud quality gates pass before merging.

2.2 Guidelines for contributors

Coding style

Backend (Java/Spring Boot):

- Follow standard Java naming conventions (camelCase for methods/variables, PascalCase for classes)
- Use Lombok annotations (`@Builder`, `@Data`, `@RequiredArgsConstructor`) to reduce boilerplate
- Apply Spring Boot best practices for dependency injection (constructor injection preferred)
- Use meaningful variable and method names that convey intent
- Maximum line length: 120 characters
- Use `@DisplayName` in tests for human-readable test descriptions

Frontend (React/TypeScript):

- Use Biome for linting and formatting (enforced via `pnpm check`)
- Follow functional component patterns with React hooks
- Use TypeScript strict mode - avoid `any` types
- Component files use `.tsx`, utility files use `.ts`
- Use TanStack Router conventions for route definitions

Code reviewing

- All code changes require a Pull request with at least one approving review
- Use the PR template (.github/PULL_REQUEST_TEMPLATE.md) which includes testing checklists
- Reviews should check:
    - Test coverage for new functionality
    - Adherence to coding standards
    - Security implications
    - Performance considerations
- Reviewers should run the branch locally for significant UI changes

2.3 Code quality metrics and dashboards

Static analysis Tool: SonarCloud

Quality gates defined:

Metric	**Threshold**	**Rationale**
Line coverage (backend)	>= 80%	Ensures comprehensive test coverage for business logic
Branch Coverage (Backend)	≥ 70%	Validates conditional logic is tested
Per-Class Line Coverage	≥ 50%	Catches completely untested classes
Code Smells	Grade A	Maintains clean, maintainable code
Security Hotspots	0	Ensures no unreviewed security issues

JaCoCo Configuration:Coverage is enforced during `mvn verify` with the following exclusions (non-business logic):

- Configuration classes (`config/*`)
- Data transfer objects (`dto/*`)
- Entity models (`model/*`)
- Application entry point

Frontend Coverage:

- Vitest generates LCOV reports uploaded to SonarCloud
- Test exclusions: test files, generated route tree

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

1. Story Assignment: Developer picks a user story from the JIRA backlog (project key: SCRUM)
2. Branch Creation: Create a feature branch following the pattern: `feature/SCRUM-XXX-short-description`
3. Development: Implement the feature with accompanying tests
4. Local Verification: Run `mvn verify` (backend) and `pnpm check && pnpm test` (frontend)
5. Pull Request: Open PR against `master` with completed PR template checklist
6. CI Validation: Wait for all CI checks to pass (tests, lint, SonarCloud)
7. Code Review: Obtain approval from at least one team member
8. Merge: Squash-merge to `master`
9. Workflow Model: GitHub Flow (single `master` branch, feature branches for development)

Definition of done

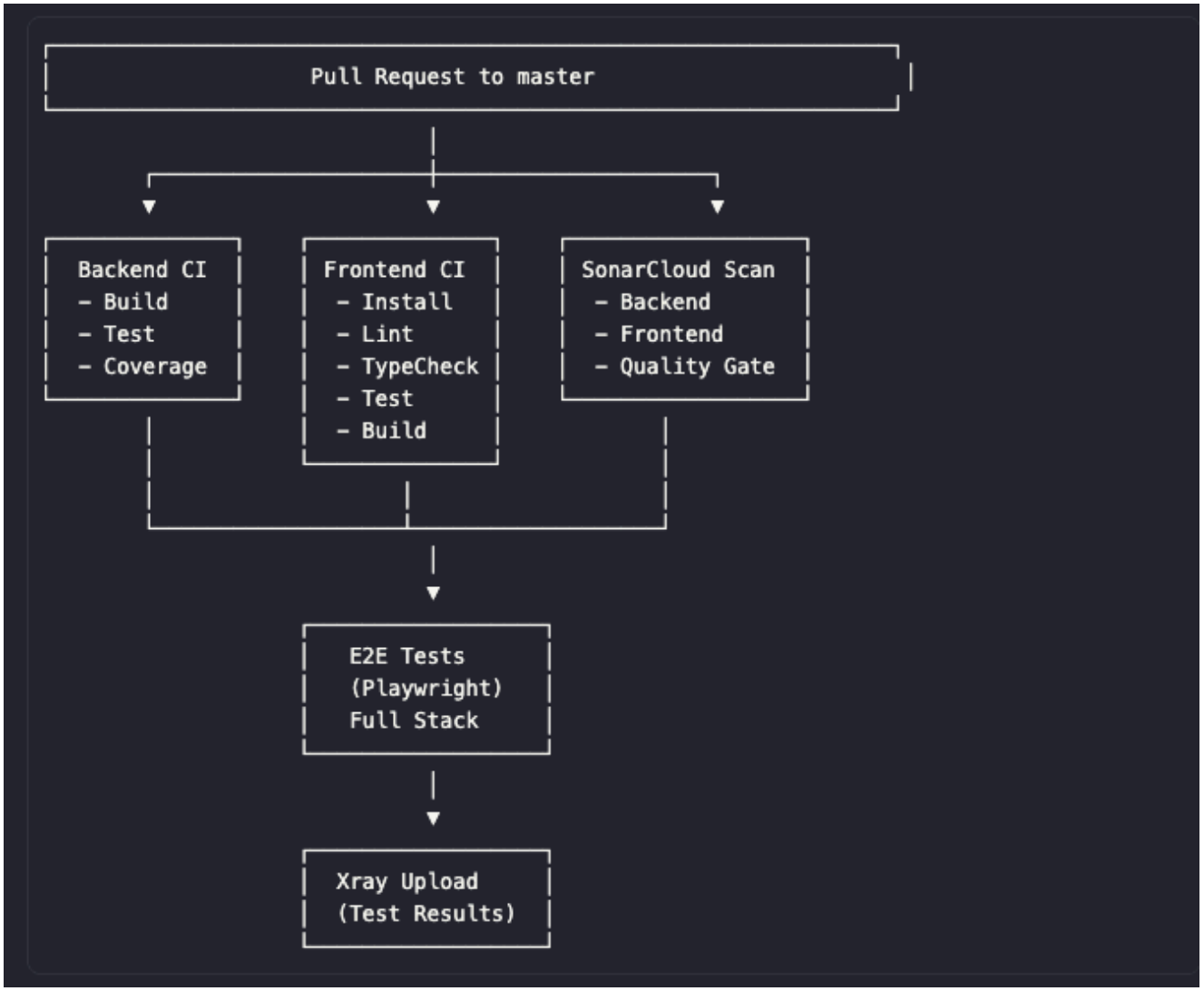
A user story is considered Done when:

- [ ] All acceptance criteria are met
- [ ] Gherkin scenarios are written and passing (if applicable)
- [ ] Unit tests written for business logic (services)
- [ ] Integration tests written for API endpoints (if applicable)
- [ ] Code coverage thresholds are maintained
- [ ] SonarCloud quality gate passes
- [ ] Code reviewed and approved
- [ ] E2E tests pass (for user-facing features)
- [ ] Tests linked to story in Xray
- [ ] Documentation updated (if API changes)
- [ ] Feature demonstrated to stakeholder

3.2 CI/CD pipeline and tools

CI/CD Platform: GitHub Actions

Pipeline Overview



Backend CI (`backend-ci.yml`)

- Build -> Maven -> Compile Java 25 with Spring Boot 4.0
- Unit Tests -> JUnit, Mockito -> Run *Test.java
- Integration Tests -> Rest Assured -> Run *IT.java files via Failsafe
- BDD Tests -> Cucumber -> Execute Gherkin scenarios
- Coverage -> JaCoCo -> Enforce >= 80% line, >=70% branch coverage
- Results Upload -> Xray -> Upload JUnit and Cucumber reports

Frontend CI (`backend-ci.yml`)

- Build -> pnpm 10 -> Install dependencies
- Lint + Format -> Biome -> Enforce code style
- Type Check -> TypeScript -> Validate type safety
- Unit Tests -> Vitest -> Component and utility tests with coverage
- Buid -> Vite 7 -> Production build version

E2E Tests

- Tool: Playwright with Chromium
- Environment: Docker Compose (docker-compose.e2e.yml) with PostgreSQL, Backend, Frontend

- Execution: Runs after Backend/Frontend/SonarCloud jobs pass
- Reports: Playwright HTML reports uploaded as artifacts

Continuous Deployment (`cd.yml`)

Triggered on merge to `master`:

1. **Change Detection: Identify modified paths (backend/frontend/migrations)**
1. **Database Migrations: Run Flyway migrations against production PostgreSQL**
1. **Backend Deploy: Trigger Render webhook for backend service**
1. **Frontend Deploy: Trigger Render webhook for static site**

Environments:

- **Development: Local Docker Compose**
- **CI: GitHub Actions runners with Testcontainers**
- **Production: Render.com (PostgreSQL + Web Services)**

3.3 System observability

Health Endpoint: `GET /api/health`

- Returns application health status
- Used by Docker healthchecks and load balancers

Monitoring Capabilities:

- Spring Boot Actuator endpoints enabled
- Render.com provides:
- Service health monitoring
- Automatic restart on failure
- Deploy logs and history

Database Observability:

- Flyway migration history tracked in `flyway_schema_history` table
- Migration validation runs before each deployment

CI/CD Observability:

- GitHub Actions provides workflow run history
- Test artifacts retained for 7 days:
  - JaCoCo coverage reports
  - Playwright test reports and screenshots
  - Cucumber HTML reports

3.4 Artifacts repository [Optional]

Test Reports:

- Backend: `backend/target/site/jacoco/` (JaCoCo), `backend/target/cucumber-reports/` (Cucumber)
- Frontend: `frontend/coverage/` (Vitest LCOV)
- E2E: `frontend/playwright-report/` (Playwright HTML)

Build Artifacts:

- Frontend production builds uploaded to GitHub Actions artifacts
- Docker images built on-demand by Render.com

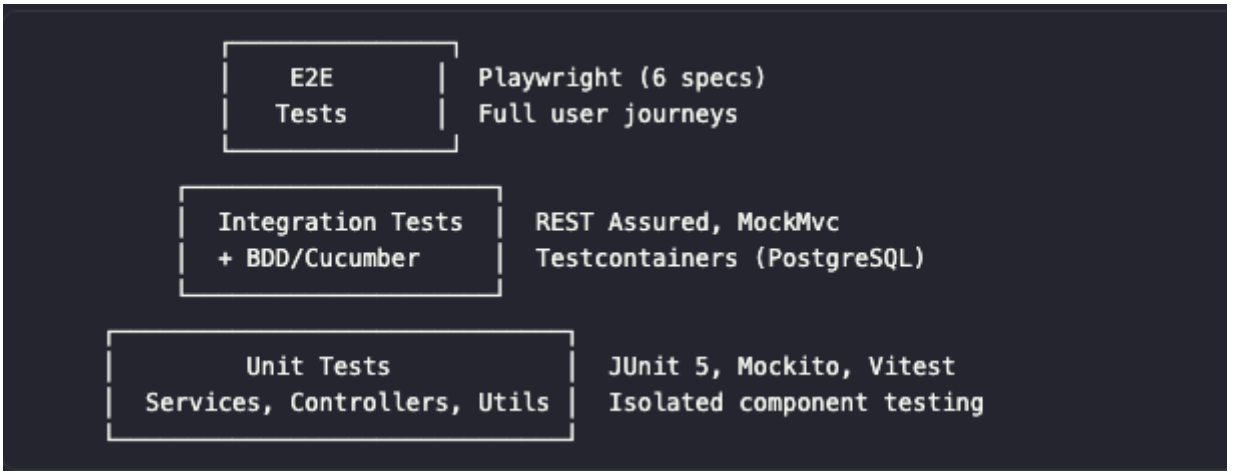
Maven Dependencies:

- Cached in GitHub Actions using `actions/cache`
- Local `.m2/repository` persisted in Docker volumes for development

4 Continuous testing

4.1 Overall testing strategy

The project employs a multi-layered testing strategy aligned with the testing pyramid:



Testing Philosophy:

- TDD/BDD Aproach: Write Gherkin scenarios for acceptance criteria, then implement tests
- Test Early, Test Often: Test run on every PR via CI
- Coverage Enforcement: Build fails if coverage drops below thresholds

4.2 Acceptance testing and ATDD

Tool: Cucumber with Gherkin syntax

Feature Files location: backend/src/test/resources/features

Gherkin Example (Login):

Feature: User Login

As a registered user

I want to login with my credentials

So that I can access the platform features

Scenario: Successful login

Given I am a registered user with email "user@ua.pt" and password "SecurePass123"

When I submit login with email "user@ua.pt" and password "SecurePass123"

Then the response status should be 200

And the response should include my user details

And the response should include a valid JWT token

When to Write Acceptance Tests:

- When implementing a new user story with clear acceptance criteria
- When fixing a bug that affects user-facing behavior
- When adding new API endpoints that serve business features

4.3 Developer facing tests (unit, integration)

Unit tests

Backend:

- Framework: JUnit 5 with Mockito for mocking
- Naming Convention: *Test.java
- Location: backend/src/test/java/tqs
- Goal: test components in isolation
- Test categories:
  - Service tests -> Business logic isolation
  - Controller tests -> Request/response mapping
  - Exception Handler -> Error response formatting

Frontend:

- Framework: Vitest with React Testing Library
- Naming Convention: *.spec.ts / *.spec.tsx
- Location: frontend/src/test/
- Coverage:

    - Route tests
    - Component Tests
    - Hook/Context Tests
    - Library Tests

Integration Tests

Backend:

- Framework: Spring Boot Test with MockMvc, Rest Assured
- Naming Convention: *IT.java
- Database: H2 in-memory
- Examples:
    - AuthControllerIT -> Authentication flow with JWT
    - OpportunityControllerIT -> Full CRUD with authorization

API Testing

Tool: Rest Assured (integrated with Cucumber steps)

4.4 Exploratory testing

Approach: Ad-hoc exploratory testing is performed during development and before releases.

Focus Areas:

- UI/UX flows not covered by automated E2E tests
- Edge cases in form validation
- Responsive design across screens sizes
- Browser compatibility

Documentation:

- Critical bugs found during exploratory testing are documented as JIRA issues
- Recurring issues are converted to automated tests

4.5 Non-function and architecture attributes testing

Performance Considerations

Current Measures:

- Pagination implemented for list endpoints
- Database indexes on frequently queried columns
- Connection pooling with Spring Boot

Load Testing:

- Not currently automated in CI

Security Testing

Implemented Security Features:

- JWT-based authentication with role-based authorization
- Password hashing with BCrypt
- CORS configuration for frontend origin
- Input validation on all endpoints

Architecture Validation

Database Schema:

- Flyway migrations ensure schema consistency
- Migration validation runs in CI and before production deploys