

TQS: Quality Assurance Manual

SportsLink

Sports Facility Rental Platform



Authors:

Paulo Cunha	Team Coordinator
Tomás Hilário	Product Owner
Rafael Ferreira	QA Engineer
Diogo Duarte	DevOps Master

Version: December 2025

University of Aveiro
Software Testing and Quality
(TQS)

Contents

Introduction	3
1 Project Management	4
1.1 Assigned Roles	4
1.1.1 Leadership Practices	4
1.2 Backlog Grooming and Progress Monitoring	5
1.2.1 Jira Organization	5
1.2.2 Refinement Routine	5
1.2.3 Ready for Development Criteria	5
1.2.4 Progress Monitoring	5
1.2.5 XRay Test Management	6
2 Code Quality Management	7
2.1 Team Policy for the Use of Generative AI	7
2.1.1 Policy Objectives	7
2.1.2 Permitted Practices (DOs)	7
2.1.3 Prohibited Practices (DON'Ts)	8
2.1.4 Responsibility for AI-Generated Code	8
2.2 Guidelines for Contributors	8
2.2.1 Code Style	8
2.2.2 Code Review Policy	9
2.3 Code Quality Metrics and Dashboards	10
2.3.1 Static Analysis Tools	10
2.3.2 Quality Gate	10
2.3.3 Monitored Metrics	10
2.3.4 Quality Dashboards	11
3 Continuous Delivery Pipeline (CI/CD)	12
3.1 Development Workflow	12
3.1.1 User Story Selection	12
3.1.2 Git Branching Strategy	12
3.1.3 Code Review in Workflow	12
3.1.4 Definition of Done (DoD)	13
3.2 CI/CD Pipeline and Tools	13
3.2.1 Tools and Technologies	13
3.2.2 CI Pipeline Stages	14
3.2.3 Test Execution Order	14
3.2.4 XRay Integration	15
3.2.5 Continuous Deployment	15
3.2.6 Environments	15
3.3 System Observability	15
3.3.1 Logging	15
3.3.2 Metrics and Monitoring	16
3.3.3 Alerts and Notifications	16
3.4 Artifacts Repository	16
3.4.1 Maven Repository	16
3.4.2 CI Artifacts	16

4	Software Testing (Continuous Testing)	17
4.1	Overall Testing Strategy	17
4.1.1	Testing Frameworks	17
4.1.2	CI/CD Integration	17
4.2	Functional Testing and ATDD	18
4.2.1	Acceptance Test-Driven Development	18
4.2.2	Acceptance Test Policy	18
4.2.3	Selenium WebDriver Configuration	18
4.3	Developer-Facing Tests (Unit, Integration)	19
4.3.1	Unit Testing Policy	19
4.3.2	Integration Testing Policy	19
4.3.3	Test Organization	20
4.4	Exploratory Testing	20
4.4.1	Strategy	20
4.5	Non-Functional and Architecture Testing	20
4.5.1	Performance Testing	20
4.5.2	Security Testing	21
4.5.3	Architecture Testing	21
A	Appendix	22
A.1	Quality Gate Summary	22
A.2	Tool Reference	22
A.3	Document Revision History	22

Introduction

This document defines the Quality Assurance (QA) policies, practices, and processes adopted by the SportsLink development team. It serves as a comprehensive guide for all team members and stakeholders to understand how quality is ensured throughout the software development lifecycle.

SportsLink is a digital marketplace platform that connects field owners with sports enthusiasts who want to rent sports fields and related equipment on-demand. The platform addresses a critical inefficiency in sports facility utilization by enabling:

- **Field Owners:** Monetization of underutilized sports facilities
- **Renters:** Easy discovery and reservation of nearby sports fields
- **Administrators:** Platform health monitoring and user management

Technology Stack:

- **Backend:** Spring Boot 3.5.7, Java 21
- **Database:** PostgreSQL 16
- **Storage:** MinIO (S3-compatible)
- **Authentication:** JWT with Spring Security
- **Payments:** Stripe API
- **Deployment:** Docker Compose

Scope of this document: This manual focuses on policies, practices, and tool configuration evidence (CI, static analysis, and test management). It does not include full test contents; instead, it defines how tests are created, executed, and used to gate quality throughout the delivery pipeline.

1 Project Management

1.1 Assigned Roles

The SportsLink team follows an Agile methodology with clearly defined roles to ensure efficient project delivery and quality assurance.

Table 1: Team Role Assignment

Role	Team Member	Key Responsibilities
Team Coordinator	Paulo Cunha	Sprint planning and coordination Team communication facilitation Impediment removal Progress tracking XRay test management
Product Owner	Tomás Hilário	Backlog prioritization Stakeholder communication Acceptance criteria definition User story creation
QA Engineer	Rafael Ferreira	Test strategy definition Test automation implementation Quality metrics monitoring
DevOps Master	Diogo Duarte	CI/CD pipeline maintenance Infrastructure management SonarQube configuration Deployment automation

1.1.1 Leadership Practices

- **Servant Leadership:** Team Coordinator removes blockers and facilitates team productivity
- **Collaborative Decision-Making:** Technical decisions are discussed in team meetings
- **Transparency:** All work is visible through Jira dashboards
- **Continuous Improvement:** Regular retrospectives to identify process improvements

1.2 Backlog Grooming and Progress Monitoring

1.2.1 Jira Organization

The team uses **Jira** as the primary project management tool with the following structure:

- **Project Key:** SL (SportsLink)
- **Issue Types:** Epic, Story, Task, Bug, Sub-task
- **Sprints:** 1-week iterations
- **Boards:** Scrum board with customized columns (To Do, In Progress, Code Review, Testing, Done)

Rationale: We use 1-week sprints to shorten feedback cycles and keep scope small, which reduces integration risk and helps maintain continuous quality control.

1.2.2 Refinement Routine

Table 2: Backlog Grooming Schedule

Activity	Frequency
Backlog Refinement Session	Mid-Week (mid-sprint)
Sprint Planning	Start of each sprint
Daily Stand-up	Daily (15 minutes)
Sprint Review	End of each sprint
Sprint Retrospective	End of each sprint

1.2.3 Ready for Development Criteria

A user story is considered **Ready for Development** when:

- Clear and concise description following user story format
- Acceptance criteria defined and testable
- Story points estimated by the team
- Dependencies identified and resolved
- Technical approach discussed and agreed
- No blocking questions remain

1.2.4 Progress Monitoring

Metrics Tracked:

- **Velocity:** Story points completed per sprint
- **Burndown Chart:** Real-time sprint progress visualization
- **Cumulative Flow Diagram:** Work in progress monitoring
- **Lead Time:** Time from story creation to completion

1.2.5 XRay Test Management

XRay is integrated with Jira for comprehensive test management. The initial configuration of XRay in Jira was performed following the guidelines provided in the theoretical lectures, adapting them to the specific needs of the SportsLink project.

XRay is integrated with Jira for comprehensive test management:

- **Test Plans:** Organized by test type:
 - **SL-103** – Unit Tests (via `maven-surefire-plugin`)
 - **SL-102** – Integration Tests (via `maven-failsafe-plugin`)
 - **SL-104** – BDD/Cucumber Tests (functional Selenium tests)
- **Test Executions:** Automatically created from CI/CD pipeline on each push to main or dev branches
- **Traceability:** Tests linked to user stories via `@Requirement("SL-XX")` annotations in Java tests and `@REQ_SL-XX` tags in Cucumber feature files to provide coverage
- **Automated Import:** JUnit XML results imported via `mikepenz/xray-action` GitHub Action, with authentication through Xray Cloud API credentials stored as repository secrets

The GitHub Actions workflow (`xray.yml`) orchestrates the following steps:

1. Setup of Java 21, Chrome browser, and PostgreSQL database
2. Execution of tests via `mvn clean verify`
3. Import of unit test results to Test Plan SL-103
4. Import of integration test results to Test Plan SL-102

To enable requirement linking in JUnit 5, the `xray-junit-extensions` dependency was added and extension autodetection was enabled via `junit-platform.properties`:

```
junit.jupiter.extensions.autodetection.enabled=true
```

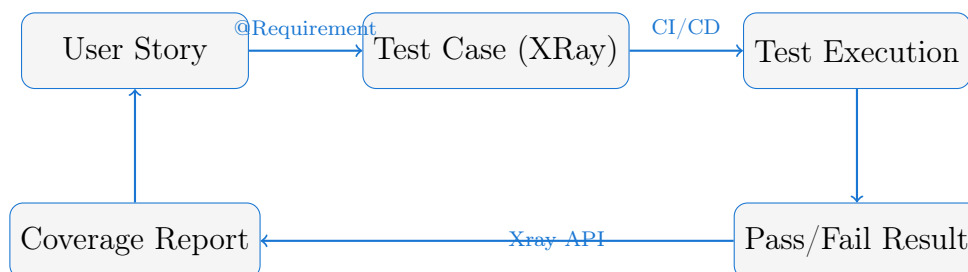


Figure 1: XRay Traceability Flow

Requirements-level coverage monitoring: By linking tests to Jira user stories (e.g., SL-XX) and importing executions automatically, the team can proactively monitor requirements coverage through XRay dashboards and identify uncovered stories early.

2 Code Quality Management

2.1 Team Policy for the Use of Generative AI

The use of generative AI tools (ChatGPT, GitHub Copilot, Claude, etc.) is **permitted** in the project, subject to strict quality and security guidelines. The goal is to enhance productivity without compromising code integrity or data security.

2.1.1 Policy Objectives

- Increase efficiency in writing code, documentation, and tests
- Ensure all AI-generated content meets defined quality standards
- Protect confidentiality of project and client data
- Ensure developers maintain complete understanding of produced code

2.1.2 Permitted Practices (DOs)

Category	Allowed Uses
Code Generation	<ul style="list-style-type: none">• Generate boilerplate code or initial examples• Explore implementation alternatives• Refactoring suggestions
Testing	<ul style="list-style-type: none">• Suggest test scenarios and edge cases• Generate JUnit, Mockito, RestAssured, Selenium, k6 tests• QA must validate all generated tests
Documentation	<ul style="list-style-type: none">• Help write technical documentation• Explain concepts, patterns, and frameworks
Code Review	<ul style="list-style-type: none">• Informal quality review• Identify potential risks• Suggest improvements

Table 3: Permitted AI Usage

2.1.3 Prohibited Practices (DON'Ts)

Category	Prohibited Actions
Sensitive Data	<ul style="list-style-type: none"> • NEVER share user or client data • NEVER share credentials, tokens, or secrets • NEVER share infrastructure information
Code Sharing	<ul style="list-style-type: none"> • NEVER share complete sensitive modules • NEVER share significant repository portions
Blind Acceptance	<ul style="list-style-type: none"> • NEVER accept AI code without full review • NEVER skip test creation/execution • NEVER bypass SonarQube validation • NEVER ignore Definition of Done

Table 4: Prohibited AI Usage

2.1.4 Responsibility for AI-Generated Code

- All AI-suggested code is considered **team code**
- **Final responsibility** for quality, security, and integration is **always human**
- Pull requests with AI-generated code must follow:
 - Project PR checklist
 - SonarQube quality gates
 - Peer review validation

2.2 Guidelines for Contributors

- Pick a Jira story, create a **feature/*** branch, and reference the story key in commits and PR title.
- Ensure CI checks pass locally when possible; open the PR early to get fast feedback.
- For AI assistance: use it for boilerplate and ideas, but always review, test, and ensure compliance with the Definition of Done.
- Link tests to requirements using `@Requirement("SL-XX")` (JUnit) or `@REQ_SL-XX` (Cucumber) for traceability.

2.2.1 Code Style

The project follows established Java code conventions with the following specifics:

Aspect	Convention
Naming - Classes	PascalCase (e.g., <code>FacilityService</code>)
Naming - Methods	camelCase (e.g., <code>findByOwner</code>)
Naming - Constants	SCREAMING_SNAKE_CASE
Naming - Variables	camelCase
Package Structure	<code>tqs.sportslink.{boundary service data dto config util}</code>
Indentation	4 spaces (no tabs)
Line Length	Maximum 120 characters
Imports	No wildcard imports
Documentation	Javadoc for public APIs

Table 5: Code Style Conventions

Reference Guides:

- [Google Java Style Guide](#)
- [Spring Framework Best Practices](#)

2.2.2 Code Review Policy**When Pull Requests are Required:**

- All changes to `main` branch
- All feature implementations
- All bug fixes
- All configuration changes

Review Requirements:

- Minimum **2 approval** required before merge
- Reviewer cannot be the PR author
- All automated checks must pass

Review Checklist:

Code compiles without errors

Unit tests added/updated and passing

Integration tests passing

No critical SonarQube issues

Code follows project conventions

No hardcoded credentials or secrets

2.3 Code Quality Metrics and Dashboards

2.3.1 Static Analysis Tools

SonarQube Cloud is the primary static analysis tool, integrated into every commit through the CI/CD pipeline.

Tool	Purpose	Integration
SonarQube Cloud	Static code analysis	Every commit via GitHub Actions
JaCoCo	Code coverage reporting	Integrated with Maven
XRay	Functional test coverage	Jira integration

Table 6: Quality Analysis Tools

Quality gates and metrics are monitored in SonarCloud: [SonarCloud Project Dashboard](#).

2.3.2 Quality Gate

The following quality gate is enforced automatically:

Metric	Threshold	Action on Failure
Code Coverage	$\geq 80\%$	Block merge
Code Duplication	$< 5\%$	Block merge
Critical Issues	0 allowed	Block merge
Security Vulnerabilities	0 critical allowed	Block merge
Code Smells (High)	0 in new code	Warning

Table 7: Quality Gate Configuration

Rationale: The selected thresholds balance development speed with risk reduction. A minimum coverage of $\geq 80\%$ ensures critical paths are consistently protected, while duplication $< 5\%$ prevents maintainability degradation.

2.3.3 Monitored Metrics

- **Code Coverage (SonarQube):** Includes unit and integration tests. Target: $\geq 80\%$
- **User Story Coverage (XRay):** Evaluates if each story has associated tests
- **Code Duplication:** Measured by SonarQube. Target: $< 5\%$
- **Code Smells:** Maintainability indicator. Target: 0 high-severity in new code
- **Bugs & Vulnerabilities:** Automatically reported. Target: 0 critical before merge
- **Technical Debt:** Time estimated to fix all code smells

2.3.4 Quality Dashboards

SonarQube Cloud Dashboard displays:

- Coverage, Bugs, Vulnerabilities, Code Smells, Duplication
- Metrics per module, branch, commit, and PR
- Technical debt estimation
- Quality gate status

XRay Dashboard (Jira) displays:

- Test execution rate
- Failures per sprint
- Requirements coverage
- Regression results

3 Continuous Delivery Pipeline (CI/CD)

3.1 Development Workflow

3.1.1 User Story Selection

Developers select work following this process:

1. Review sprint backlog in Jira
2. Select highest priority unassigned story
3. Assign story to self and move to “In Progress”
4. Create feature branch from `dev`
5. Implement feature following DoD
6. Create Pull Request for review

3.1.2 Git Branching Strategy

The project follows a **simplified GitHub Flow**:

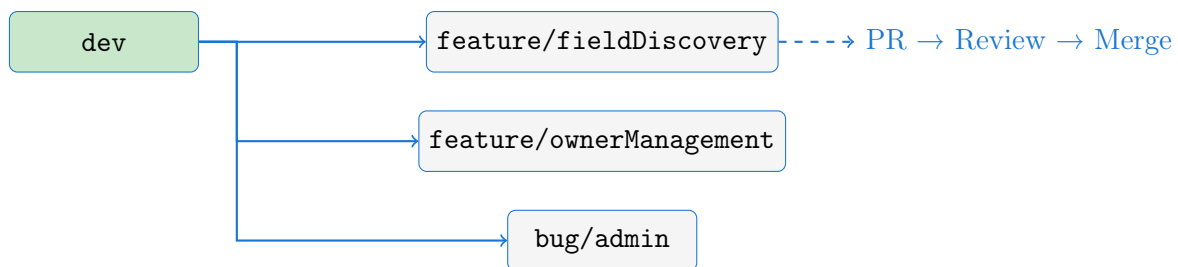


Figure 2: Git Branching Strategy

Branch Naming Convention:

- Features: `feature/short-description`
- Bugfixes: `bug/short-description`

Rationale: We adopted a simplified GitHub Flow to minimize long-lived branches and reduce merge conflicts, ensuring fast feedback through pull requests and CI checks.

3.1.3 Code Review in Workflow

1. Developer completes implementation and pushes to feature branch
2. Developer creates Pull Request with description
3. CI pipeline runs automatically (tests, analysis)
4. Reviewer(s) assigned automatically or manually

5. Review comments addressed by developer
6. Upon approval and passing checks, PR is merged
7. Feature branch is deleted after merge

3.1.4 Definition of Done (DoD)

A user story is considered **Done** when:

- All acceptance criteria validated
- Unit tests implemented and passing
- Integration tests implemented (if applicable) and passing
- Minimum code coverage target met ($\geq 80\%$)
- No critical static analysis findings
- Code review completed and approved
- Merged to `main` branch
- Deployed to production environment

3.2 CI/CD Pipeline and Tools

3.2.1 Tools and Technologies

Tool	Purpose	Configuration
GitHub Actions	CI/CD orchestration	<code>.github/workflows/ci.yml</code>
Maven	Build and dependency management	<code>pom.xml</code>
JaCoCo	Code coverage	Maven plugin
SonarQube Cloud	Static analysis	Cloud-hosted
Docker Compose	Containerization	<code>docker-compose.yml</code>
Self-hosted Runner	Deployment to VM	GitHub Actions runner

Table 8: CI/CD Tool Stack

Evidence: The CI workflow is defined in [.github/workflows/ci.yml](#) in the SportsLink repository.

3.2.2 CI Pipeline Stages

The continuous integration pipeline executes on every push on main and pull request and merges:

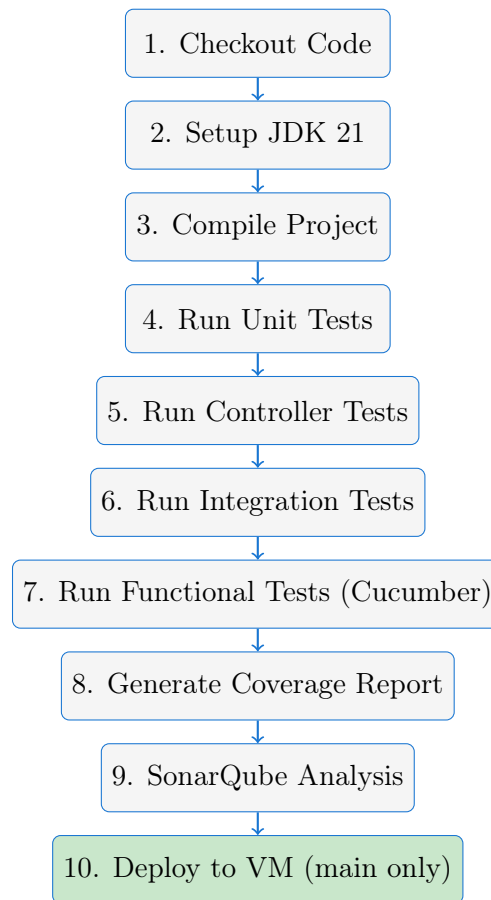


Figure 3: CI/CD Pipeline Stages

3.2.3 Test Execution Order

Tests are organized in layers and executed sequentially:

Listing 1: Test Execution in CI Pipeline

```

1 # 1. Repository Layer Tests (DataJpaTest)
2 mvn test -Dtest=**/A_Tests_repository/*Test
3
4 # 2. Service Layer Unit Tests
5 mvn test -Dtest=**/B_Tests_unit/*Test
6
7 # 3. Controller Tests (MockMvc)
8 mvn test -Dtest=**/C_Tests_controller/*Test
9
10 # 4. Integration Tests (RestAssured)
11 mvn test -Dtest=**/D_Tests_integration/*Test
12
13 # 5. Functional Tests (Cucumber + Selenium)
14 mvn test -Dtest=RunCucumberTest
  
```

3.2.4 XRay Integration

A separate workflow (`xray.yml`) imports test results to XRay:

- **Unit Tests:** Imported as JUnit format to Test Plan SL-103
- **Integration Tests:** Imported as JUnit format to Test Plan SL-102
- **BDD Tests:** Imported as Cucumber JSON format

The XRay import workflow is defined in [.github/workflows/xray.yml](#).

3.2.5 Continuous Deployment

Deployment occurs automatically when:

- Push to `main` branch
- All CI checks pass

Deployment Process:

Listing 2: Deployment Commands

```
1 cd sportslink
2 docker compose down
3 docker compose up -d --build
4 docker image prune -f
```

3.2.6 Environments

Environment	Purpose	Database	Trigger
Development	Local testing	H2 (in-memory)	Manual
CI/Testing	Automated tests	PostgreSQL (container)	Every push
Production	Live application	PostgreSQL (persistent)	Merge to main

Table 9: Environment Configuration

3.3 System Observability

3.3.1 Logging

Aspect	Configuration
Framework	SLF4J with Logback (Spring Boot default)
Log Levels	ERROR, WARN, INFO, DEBUG
Output	Console + File (<code>app.log</code>)
Format	Timestamp, Level, Thread, Logger, Message
Retention	7 days (CI artifacts)

Table 10: Logging Configuration

3.3.2 Metrics and Monitoring

- **Application Metrics:** Spring Boot Actuator endpoints
- **Container Metrics:** Docker stats and logs
- **Database Metrics:** PostgreSQL built-in statistics
- **CI/CD Metrics:** GitHub Actions workflow insights

3.3.3 Alerts and Notifications

- **CI Failures:** GitHub notifications to team members
- **Quality Gate Failures:** SonarQube email notifications
- **Deployment Status:** GitHub Actions status checks

3.4 Artifacts Repository

3.4.1 Maven Repository

- **Dependency Source:** Maven Central
- **Local Cache:** ~/.m2 (cached in CI)
- **Build Artifacts:** Stored in `target/` directory

3.4.2 CI Artifacts

GitHub Actions stores test reports and logs:

Artifact	Path	Retention
Surefire Reports	<code>target/surefire-reports/</code>	30 days
Cucumber Reports	<code>target/cucumber-reports/</code>	30 days
Application Logs	<code>app.log</code>	7 days

Table 11: CI Artifact Storage

4 Software Testing (Continuous Testing)

4.1 Overall Testing Strategy

The SportsLink testing strategy follows **Test-Driven Development (TDD)** principles with a multi-layered approach ensuring continuous validation at every stage.

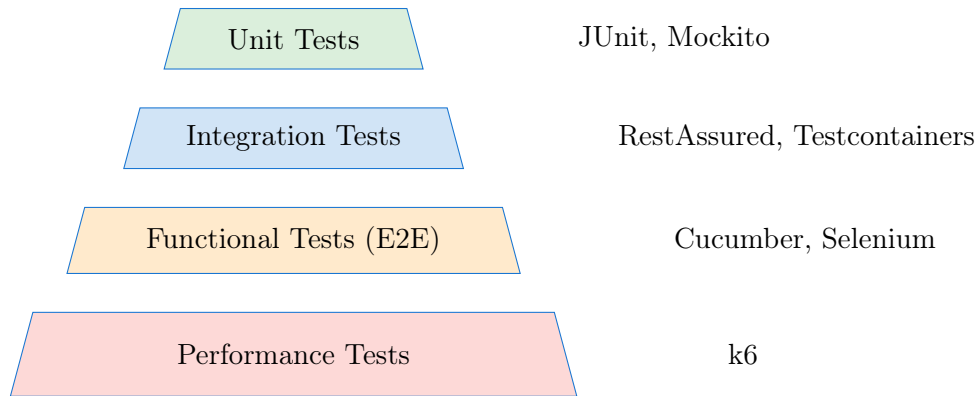


Figure 4: Testing Pyramid

Alignment with TQS practices: The team follows TDD for core service logic (unit tests first for business rules) and ATDD/BDD for user-facing acceptance criteria using Cucumber scenarios, ensuring that tests drive development and remain traceable to requirements.

4.1.1 Testing Frameworks

Test Type	Framework	Purpose	Execution
Unit	JUnit 5, Mockito	Isolated logic validation	Every push
Integration	RestAssured, Testcontainers	API and database testing	Every push
Functional (E2E)	Cucumber, Selenium	User flow validation	Every push
Performance	k6	Load and stress testing	On demand

Table 12: Testing Framework Stack

4.1.2 CI/CD Integration

- All tests run automatically on every push and pull request
- Test results are uploaded to XRay for traceability
- Failed tests block merge to `main` branch
- Coverage reports are generated and published to SonarQube

4.2 Functional Testing and ATDD

4.2.1 Acceptance Test-Driven Development

Functional tests follow the **Behavior-Driven Development (BDD)** methodology using Cucumber:

Listing 3: Example Cucumber Feature

```

1  # =====
2  # SEARCH SCENARIOS
3  # User Story: SL-27 (Search by name, sport type)
4  # Epic: SL-15 (Field Discovery & Catalog - se existir)
5  # =====
6
7  @SL-27 @SL-104 @search @functional
8  Scenario: User searches for fields using the sport dropdown
9      Given I am on the main search page
10     When I select the sport "Football" from the dropdown
11     And I press the search button
12     Then I should see facilities related to "Football"
13
14  @SL-27 @SL-104 @search @functional
15  Scenario: User searches for padel courts in Aveiro
16     Given I am on the main search page
17     When I select the sport "Padel" from the dropdown
18     And I enter "Aveiro" into the location field
19     And I press the search button
20     Then I should see facilities located in "Aveiro"

```

4.2.2 Acceptance Test Policy

- **Who writes:** QA Engineer in collaboration with Product Owner
- **When written:** During sprint planning, before implementation
- **Execution:** Automated with Selenium WebDriver
- **Traceability:** Mapped to user stories in XRay
- **Coverage:** Each user story must have at least one acceptance test

4.2.3 Selenium WebDriver Configuration

- Browser: Google Chrome (headless in CI)
- WebDriver Manager: Selenium-Jupiter for automatic driver management
- Wait Strategy: Explicit waits for element visibility
- Screenshots: Captured on test failure

4.3 Developer-Facing Tests (Unit, Integration)

4.3.1 Unit Testing Policy

When to Write Unit Tests:

- For all service layer methods
- For complex business logic
- For utility classes and helpers
- For data validation logic

Unit Test Requirements:

- Use **JUnit 5** and **Mockito** for mocking dependencies
- Cover positive cases, negative cases, and edge cases
- Test methods should be independent and isolated
- Follow Arrange-Act-Assert (AAA) pattern
- Naming convention: `methodName_scenario_expectedResult`

Example Scenarios:

- `createRental_validData_returnsRentalDTO`
- `createRental_overlappingTime_throwsConflictException`
- `calculatePrice_multipleEquipments_sumsPricesCorrectly`

4.3.2 Integration Testing Policy

When to Write Integration Tests:

- For all REST API endpoints
- For database operations (repositories)
- For external service integrations
- For authentication and authorization flows

Integration Test Requirements:

- Use **RestAssured** for API testing
- Use **Testcontainers** or embedded databases
- Test complete request/response cycle
- Verify HTTP status codes, headers, and body
- Test error handling and edge cases

Example Scenarios:

- `POST /api/auth/login` with valid credentials returns JWT token
- `GET /api/facilities` returns paginated list of facilities
- `POST /api/rentals` with invalid time slot returns 409 Conflict

4.3.3 Test Organization

Tests are organized in separate packages for clear execution:

Listing 4: Test Package Structure

```

1 src/test/java/tqs/sportslink/
2   A_Tests_repository/      # Repository/DataJpaTest tests
3   B_Tests_unit/           # Service layer unit tests
4   C_Tests_controller/     # Controller tests with MockMvc
5   D_Tests_integration/    # Full API integration tests
6   functionals/            # Cucumber step definitions

```

4.4 Exploratory Testing

4.4.1 Strategy

Exploratory testing is performed **manually each sprint** to discover unexpected behaviors, especially in new or sensitive features.

Focus Areas:

- New feature implementations
- Complex user workflows
- Edge cases not covered by automated tests
- UI/UX inconsistencies
- Cross-browser compatibility

Process:

1. QA Engineer identifies high-risk areas
2. Time-boxed exploration sessions (1-2 hours)
3. Findings documented in Jira as bugs or improvements
4. Critical bugs added to automated regression suite

4.5 Non-Functional and Architecture Testing

4.5.1 Performance Testing

Tool: k6 (load testing framework)

Test Types:

- **Load Testing:** Normal expected load
- **Stress Testing:** Beyond normal capacity
- **Spike Testing:** Sudden traffic increases

Metrics Monitored:

- Response time (p50, p95, p99)
- Throughput (requests per second)
- Error rate
- Resource utilization (CPU, memory)

4.5.2 Security Testing

- **Static Analysis:** SonarQube security hotspots detection
- **Authentication Testing:** JWT token validation, expiration
- **Authorization Testing:** Role-based access control verification
- **Input Validation:** SQL injection, XSS prevention

4.5.3 Architecture Testing

- **Layered Architecture:** Verified through package structure
- **Dependency Injection:** Spring IoC container
- **API Contracts:** RESTful conventions followed
- **Database Integrity:** Foreign key constraints, transactions

A Appendix

A.1 Quality Gate Summary

Metric	Threshold	Status
Code Coverage	$\geq 80\%$	Enforced
Code Duplication	$< 5\%$	Enforced
Critical Issues	0	Enforced
Security Vulnerabilities	0 critical	Enforced

Table 13: Quality Gate Summary

A.2 Tool Reference

Category	Tool	URL/Reference
CI/CD	GitHub Actions	https://github.com/features/actions
Static Analysis	SonarQube Cloud	https://sonarcloud.io
Test Management	XRay (Jira)	https://www.getxray.app
Unit Testing	JUnit 5	https://junit.org/junit5
Mocking	Mockito	https://site.mockito.org
API Testing	RestAssured	https://rest-assured.io
BDD	Cucumber	https://cucumber.io
Browser Automation	Selenium	https://www.selenium.dev
Performance	k6	https://k6.io
Containers	Docker	https://www.docker.com
Database	PostgreSQL	https://www.postgresql.org

Table 14: Tool Reference Guide

A.3 Document Revision History

Version	Date	Author	Changes
1.0	December 2025	SportsLink Team	Initial version

Table 15: Revision History