TQS: Product specification report

Pedro Rafael Lemos Rodrigues [102432]
v2026-01-30

1 Introduction

1.1 Overview of the project

This project was developed in the scope of the **Teste e Qualidade de Software (TQS)** course. The main objective of the assignment is to design, implement, and validate a software system while applying software quality assurance practices, including automated testing, static analysis, and continuous integration.

The developed application is a **volunteer management platform** that supports the publication and management of volunteer opportunities and the enrollment of volunteers. The system is intended to support organizations and users by providing a centralized platform to manage volunteer activities.

The application allows:

- Organizations to publish volunteer opportunities
- Volunteers to browse available opportunities
- Volunteers to enroll in opportunities
- Management of basic participation and opportunity data

1.2 Project limitations and know issues

Several limitations were identified during the development of the project:

- End-to-end (E2E) automated testing using Selenium was attempted but not fully completed due to technical integration complexity and time constraints.
- No performance, load, or stress testing was implemented.
- No advanced monitoring or observability mechanisms were configured.
- Authentication and authorization mechanisms are limited and not production-grade.
- The frontend-backend integration relies on basic configuration and does not include advanced deployment or scaling strategies.

These limitations were accepted in order to prioritize core functionality, automated unit and integration testing, and quality assurance practices within the scope of the course.

1.3 References and resources

Key technologies, frameworks, and resources used in the project include:

- Spring Boot (backend framework)
- React (frontend framework)
- JUnit (unit and integration testing)
- Mockito (mocking framework)
- JaCoCo (code coverage)
- SonarCloud (static analysis and quality gates)
- GitHub (source code management and CI)
- GitHub Actions (continuous integration)
- Swagger / OpenAPI (API documentation and manual API testing)
- Docker (containerization for local development and infrastructure components)
- Jira (project and backlog management)

These tools supported the implementation of both functional requirements and quality assurance practices.

2 Product concept and requirements

2.1 Vision statement

The vision of the system is to provide a simple and centralized platform for managing volunteer opportunities and volunteer participation.

At a high level, the system addresses the following business needs:

- Allow organizations to publish and manage volunteer opportunities
- Allow volunteers to discover and enroll in opportunities

- Provide a structured representation of volunteer activities
- Support basic lifecycle management of opportunities and participations

Key promised features include:

- Listing of volunteer opportunities
- Creation and management of opportunities
- Enrollment of volunteers in opportunities
- Visualization of basic opportunity and participation information

Some advanced features that could be expected in a production system (e.g., notifications, advanced user management, reporting, and analytics) were considered out of scope for this academic project.

The requirements were derived from the course assignment context and refined through iterative development using Jira user stories and epics.

2.2 Personas and scenarios

2.2.1.1 Persona 1: Volunteer

Name: João Silva
Age: 22
Occupation: University student
Profile: João is interested in participating in community and environmental activities. He uses web applications to discover volunteering opportunities that fit his schedule.

Goals:

- Find volunteer opportunities
- Enroll in activities easily
- Keep track of his participations

Scenario:
João accesses the platform to look for new volunteering opportunities. He browses the list of available activities, reads the descriptions, and selects an opportunity that fits his availability. He enrolls in the opportunity and later checks the platform to confirm his participation.

2.2.1.2 Persona 2: Organization Representative

Name: Maria Costa
Age: 35
Occupation: NGO Coordinator
Profile: Maria manages community initiatives and needs a simple tool to publish volunteering opportunities and manage basic information.

Goals:

- Publish volunteer opportunities
- Manage opportunity information
- Attract volunteers efficiently

Scenario:
Maria creates a new volunteer opportunity on the platform, specifying the title, description, date, and required effort. She publishes the opportunity so that volunteers can view and enroll. She later reviews the list of participations to monitor interest.

2.3 Project epics and priorities

The project was organized into epics to support incremental and structured development. Key epics include:

- **Epic 1: User and Access Management**
    - User registration and basic authentication
    - Management of user roles (Volunteer, Promoter, Admin)
    - Storage and validation of user credentials and profiles
- **Epic 2: Volunteer Opportunities Management**
    - Creation and management of volunteer opportunities
    - Definition of opportunity attributes (title, description, date, duration, points)
    - Listing and visualization of available opportunities
- **Epic 3: Volunteer Participation**
    - Enrollment of volunteers in opportunities
    - Management of participation lifecycle (e.g., pending, approved)
    - Prevention of duplicate enrollments for the same opportunity
- **Epic 4: Points and Rewards System**
    - Assignment of points for completed participations
    - Management of rewards and associated point costs
    - Activation and deactivation of available rewards
- **Epic 5: Monitoring and Transparency**
    - Visualization of participation status
    - Basic tracking of volunteer activity
    - Support for transparency in opportunity and participation management

Priorities were given first to user and access management and core opportunity management, followed by participation handling, points and rewards, and finally monitoring and transparency features.

3 Domain model

3.1.1 AppUser

Represents a system user.

Key attributes:

- id
- name
- email (unique)
- passwordHash
- roles (set of Role values)

Each user can have one or more roles, such as Volunteer, Promoter, or Admin.

3.1.2 Role

Represents the role assigned to a user.

Possible values:

- VOLUNTEER
- PROMOTER
- ADMIN

Roles define the permissions and responsibilities of each user in the system.

3.1.3 Opportunity

Represents a volunteer opportunity (domain concept already implemented in the project).

Key attributes (conceptual):

- id
- title
- description
- date
- durationHours
- points

An Opportunity can have multiple associated Participations.

3.1.4 Participation

Represents the enrollment of a volunteer in a specific opportunity.

Key attributes:

- id
- volunteer (AppUser)
- opportunity (Opportunity)
- status (e.g., PENDING, APPROVED)
- createdAt

Constraints and relationships:

- A volunteer can only enroll once per opportunity (unique constraint)
- Each Participation links exactly one volunteer to one opportunity

This entity implements the many-to-many relationship between users and opportunities.

3.1.5 Reward

Represents a reward that can be redeemed using points.

Key attributes:

- id
- name (unique)
- cost (in points)
- active (boolean)

Rewards support the points and incentives system in the platform.

3.1.6 Domain relationships (summary)

- An **AppUser** can have multiple **Participations**
- An **Opportunity** can have multiple **Participations**
- A **Participation** links one **AppUser** to one **Opportunity**
- The **Reward** entity is managed independently and is associated with the points system

This domain model supports the core business logic of volunteer management, participation tracking, and reward handling.

4 Architecture notebook

4.1 Key requirements and constrains

The architecture of the system was driven by both functional and quality-related requirements, as well as constraints imposed by the academic context of the project.

Key architectural requirements include:

- Support for multiple user roles (Volunteer, Promoter, Admin)

- Separation between frontend and backend concerns
- Exposure of domain functionality through a RESTful API
- Support for participation lifecycle management (e.g., pending, approved)
- Support for points and rewards management
- High testability to support automated unit and integration testing
- Maintainability and modularity to support incremental development

Key constraints include:

- Limited development time due to schedule
- Single-developer team, requiring strong automation and self-review
- Web-based deployment only (no mobile or desktop clients)
- No requirement for high availability or horizontal scalability
- No external third-party system integrations

Relevant architectural characteristics for this project include:

- **Maintainability** – ease of modifying and extending business logic
- **Testability** – ability to isolate and test services and controllers
- **Modularity** – clear separation of layers and concerns
- **Deployability** – ability to deploy using Docker containers
- **Simplicity** – preference for simple, understandable design over complex distributed architectures

These characteristics guided the choice of a layered, monolithic backend architecture with a separate frontend application.

4.2 Architecture view

The system follows a **layered client-server architecture**, with a clear separation between presentation, application, and domain logic.

4.2.1 Main architectural building blocks

4.2.1.1 Frontend (React Single Page Application)

- Implements the user interface
- Handles user interactions and navigation
- Communicates with the backend exclusively through REST API calls
- Displays data related to opportunities, participations, and rewards

Backend (Spring Boot Application)

The backend is organized into multiple logical layers:

- **REST Controllers**
    - Expose HTTP endpoints for domain resources
    - Map HTTP requests to application services
- **Service Layer**
    - Implements business logic
    - Enforces domain rules (e.g., participation uniqueness, status transitions, points assignment)
- **Domain Model**
    - Core business entities: AppUser, Opportunity, Participation, Reward, Role
    - Encapsulates the main business concepts
- **Repository Layer**
    - Handles persistence using Spring Data JPA
    - Abstracts database access from business logic

Architectural style

The backend adopts a **layered monolithic architecture**, which is well suited for:

- Educational projects
- Strong consistency requirements
- Simplicity and ease of testing
- Clear separation of concerns

The frontend and backend communicate using a **RESTful client-server model** over HTTP, exchanging JSON representations of domain resources.

4.2.2 Module interactions (conceptual sequence)

A typical interaction scenario (volunteer enrollment):

1. The volunteer uses the React frontend to select an opportunity
2. The frontend sends a POST request to the backend API to create a Participation
3. The REST controller receives the request and delegates to the service layer
4. The service layer validates business rules (e.g., no duplicate participation)
5. The repository layer persists the Participation entity
6. The backend returns a response to the frontend
7. The frontend updates the UI to reflect the new participation status

This interaction flow ensures that business rules are centralized in the backend and that the frontend remains a thin client.

4.3 Deployment view (production configuration)

The system is deployed using **Docker containers**, enabling reproducible and isolated environments.

A typical deployment configuration includes:

4.3.1 Frontend container

- Runs an Nginx web server
- Serves the React static application
- Exposes HTTP port 80 (mapped to a host port for local access, e.g., 5173)

Backend container

- Runs the Spring Boot application
- Exposes port 8080
- Provides REST API endpoints for frontend and developer access

Supporting containers (development/testing)

- Selenium container (used during attempted E2E testing)
- Optional database container (if configured for persistence)

Network and communication

- Containers communicate over a Docker virtual network
- The frontend container communicates with the backend using internal container hostnames and ports
- Host-to-container port mappings allow developers to access the system locally

This deployment model supports:

- Consistent development and testing environments
- Easy setup for new developers
- Isolation between frontend, backend, and supporting services

The deployment configuration prioritizes simplicity and reproducibility over production-grade scalability.

5 API for developers

The backend exposes a RESTful API that provides access to the main domain resources of the system. The API is designed following REST best practices, with resource-oriented endpoints, standard HTTP methods, and JSON-based data exchange.

The API is documented and exposed using **Swagger/OpenAPI**, providing:

- Interactive documentation
- Detailed request and response schemas
- Manual testing support for developers
- Up-to-date visibility of available endpoints

The API is organized around the following main resource groups:

5.1 Authentication and current user

5.1.1 Auth Controller

Responsible for user registration.

- `POST /api/auth/register`
  Registers a new user in the system.

Current User Controller

Provides information about the currently authenticated user.

- `GET /api/me`
  Retrieves profile information of the current user.

These endpoints support Epic 1 (User and Access Management).

5.2 Opportunity management

5.2.1 Opportunity Controller

Manages volunteer opportunities.

- `GET /api/opportunities`
  Retrieves the list of available opportunities.
- `GET /api/opportunities/{id}`
  Retrieves detailed information about a specific opportunity.
- `POST /api/opportunities`
  Creates a new volunteer opportunity.
- `PUT /api/opportunities/{id}`
  Updates an existing opportunity.
- `PATCH /api/opportunities/{id}/deactivate`
  Deactivates an opportunity without deleting it.

- `POST /api/opportunities/{id}/enroll`
  Enrolls the current user in a specific opportunity.

These endpoints support Epic 2 (Volunteer Opportunities Management) and Epic 3 (Volunteer Participation).

5.3 Participation management

5.3.1 Participation Controller

Manages the lifecycle of participations.

- `POST /api/participations/{id}/approve`
  Approves a participation.
- `POST /api/participations/{id}/reject`
  Rejects a participation.
- `POST /api/participations/{id}/cancel`
  Cancels a participation.
- `GET /api/participations/me`
  Retrieves current participations of the authenticated user.
- `GET /api/participations/me/history`
  Retrieves historical participations of the authenticated user.

These endpoints implement Epic 3 (Volunteer Participation) and support Epic 5 (Monitoring and Transparency).

5.4 Points and rewards

5.4.1 Points Controller

Manages user points.

- `GET /api/points/balance`
  Retrieves the current point balance of the authenticated user.

This endpoint supports Epic 4 (Points and Rewards System).

5.4.2 Reward Controller

Handles reward browsing and redemption.

- `GET /api/rewards`
  Retrieves the list of available rewards.
- `POST /api/rewards/{id}/redeem`
  Redeems a reward using the user's points.
- `GET /api/rewards/me/history`
  Retrieves the reward redemption history of the authenticated user.

These endpoints support Epic 4 (Points and Rewards System) and Epic 5 (Monitoring and Transparency).

5.5 Administrative endpoints

5.5.1 Admin User Controller

Manages user roles.

- `PUT /api/admin/users/{id}/roles`
  Updates roles assigned to a user.

Supports Epic 1 (User and Access Management).

5.5.2 Admin Points Rules Controller

Manages point assignment rules.

- `GET /api/admin/points/rules`
  Retrieves current point rules.
- `PUT /api/admin/points/rules`
  Updates point assignment rules.

Supports Epic 4 (Points and Rewards System).

5.5.3 Admin Reward Controller

Manages rewards.

- `GET /api/admin/rewards`
  Retrieves all rewards (including inactive).
- `POST /api/admin/rewards`
  Creates a new reward.
- `PATCH /api/admin/rewards/{id}`
  Updates an existing reward (e.g., activation status, cost, name).

Supports Epic 4 (Points and Rewards System).

5.6 API design principles

The API follows standard REST design principles:

- Resource-oriented URLs (e.g., `/api/opportunities`, `/api/participations`)
- Use of HTTP verbs to express actions
- JSON as the standard representation format
- Use of HTTP status codes to reflect operation results
- Clear separation between user-facing and administrative endpoints

Swagger/OpenAPI serves as the authoritative source of API documentation, ensuring that developers can easily understand and integrate with the system.