TQS: Product specification report

Pedro Fonseca[119264], Dinis Cunha[119316], Rafael Fernandes[118956], Francisco Silva[118716]

v2025-11-25

1 Introduction

1.1 Overview of the project

The project is developed within the curricular unit Testes e Qualidade de Software (TQS), whose objective is to apply software engineering principles oriented towards quality.

The developed application, BlackTie, consists of a rental platform for suits and formal clothing items, allowing users to offer or rent items in a secure and practical manner.

1.2 Project limitations and know issues

During the development of the BlackTie platform, some features initially planned were not implemented in the final version of the project, due to time constraints and development priorities. The main known limitations include:

Admin Dashboard: Although the backend provides complete API endpoints for platform metrics, user management, and product moderation, the frontend admin interface remains in a minimal state. The dashboard displays basic metrics but lacks advanced visualisations, trend analysis, and bulk management operations. This feature would require the development of more complex UI components and data aggregation logic, which was beyond the scope of the MVP.

Role Selection Logic: The role selection mechanism that allows users to switch between Renter and Owner roles has been implemented, but its behaviour has not been fully validated across all user scenarios. Due to limited end-to-end testing coverage for this specific flow, there is uncertainty about whether the role switching functions correctly in all edge cases, particularly regarding how it affects existing bookings and product listings when a user changes roles.

Return Confirmation: The workflow that would allow owners to confirm item returns and assess item condition before depositing refunds was not implemented. Currently, the booking lifecycle ends without formal owner confirmation, and disputes must be handled manually outside the platform. This feature would require integration of physical inspection workflows with digital confirmation mechanisms.

These limitations were considered secondary compared to the platform's essential flows, such as product discovery, booking management, payment processing, and review submissions. The mentioned features remain as potential future evolutions of the platform.

1.3 References and resources

Among the resources used in the project are:

Frontend & Design:

- **React** (https://react.dev/) was used for frontend development, with a focus on reusable components and responsive design. The official documentation was essential for clarifying doubts and implementing best practices related to component structure and state management.
- **Vite** (https://vitejs.dev/) was used as the build tool and development server for the frontend. It was selected due to its high performance and significantly faster hot-reload times when compared to Create React App.
- **Tailwind CSS** (https://tailwindcss.com/) is a utility-first CSS framework used to style the application. It enabled rapid UI development and ensured design consistency without the need to write custom CSS files.

Backend & Database:

- **Spring Boot** (https://spring.io/projects/spring-boot) was used as the foundation of the backend. Its convention-over-configuration approach facilitated rapid development and modular implementation of REST APIs.
- **PostgreSQL** (https://www.postgresql.org/) was chosen as the production database system. The official documentation and community resources, such as Stack Overflow, were helpful for resolving questions related to data modeling and SQL queries.

- **Stripe API** (https://stripe.com/docs/api) was used to simulate payment processing. Its clear documentation and code examples simplified the integration process with the application.

Testing & DevOps:

- **H2 Database** (https://www.h2database.com/) is an in-memory database used specifically for integration testing. It ensures that tests are executed in an isolated environment without impacting the production database.
- **JaCoCo** (https://www.eclemma.org/jacoco/) is a Java code coverage library configured in the build process to generate coverage reports. These reports are an important metric for evaluating the project's quality objectives.
- **Docker** (https://docs.docker.com/) was used to ensure consistent development and deployment environments. The official documentation and online tutorials were instrumental in creating and configuring the Dockerfiles.

2 Product concept and requirements

2.1 Vision statement

The system provides a secure and accessible platform for renting formal clothing, offering a practical solution for the occasional use of this type of product.

From a functional perspective, the platform is organized around **three main roles**. Owners can publish and manage formal clothing items, define availability periods, prices, and security deposits, and monitor reservations associated with their items.

Renters are able to search and filter available items, make date-based reservations, and complete payments using a sandbox payment system, allowing realistic simulation of payment flows without financial risk. After each rental, they can review both the rented items and the respective Owners.

Administrators are responsible for supervising the platform's operation. Their responsibilities include monitoring user activity, managing users and listings, and handling disputes to ensure compliance with platform rules.

The system was designed based on team discussions, emphasizing the product owner desires and analysis of similar rental platforms, with feature selection guided by usability, security, and software quality objectives

2.2 Personas and scenarios

Persona 1 – João Ribeiro (Renter)

- **Age:** 32
- **Profession:** Business Consultant
- **Location:** Porto
- **Digital Literacy:** Medium

João frequently attends conferences and corporate events where a professional appearance is required. He considers it unsustainable to purchase expensive suits that are only used a few times. He values efficiency, speed, and trust throughout the rental process.

Motivations: access elegant suits without purchasing them; a simple and efficient process.
Frustrations: incorrect sizing; limited availability.
Goals: quickly discover suitable items; complication-free delivery and return.

Persona 2 – Marta Faria (Owner)

- **Age:** 29
- **Profession:** Content Creator
- **Location:** Braga
- **Digital Literacy:** High

Marta owns an extensive wardrobe with formal items that are rarely used. She sees the platform as an opportunity to generate income while also promoting sustainable fashion practices.

Motivations: monetize unused items; help other users.
Frustrations: damage to items; delays in returns.
Goals: easy item publication; availability control; protection of her items.

Persona 3 – Pedro Duarte (Owner)

- **Age:** 35
- **Profession:** Tailor / Menswear Store Owner
- **Location:** Coimbra
- **Digital Literacy:** Medium

Pedro owns several premium suits in his store that have low sales turnover. Renting allows him to monetize inventory while maintaining controlled risk.

Motivations: monetize premium suits.

Frustrations: irresponsible renter behavior.
Goals: ensure secure rentals with reputation and trust mechanisms.

- **Persona 4 – Carolina Mendes (Admin)**
- **Age:** 38
- **Profession:** Operations Manager
- **Location:** Lisbon
- **Digital Literacy:** High

Carolina is responsible for keeping the platform secure, fair, and stable. She supervises disputes, suspicious behavior, and operational incidents.

Motivations: ensure platform security; resolve disputes impartially.
Frustrations: lack of evidence; fraud; need for manual intervention.
Goals: enforce rules, manage users, and maintain system credibility.

Scenario 1 – João rents a suit for a conference
Actor: João (Renter)
Motivation: Needs a suit for a professional event without purchasing one.

Action:

- Accesses the platform.
- Searches for suits and filters by size.
- Selects a suit listed by Pedro.
- Reviews photos and the item description.
- Reserves the item for a 7-day period.
- Completes the payment.
- Receives and uses the suit.
- Returns it within the agreed timeframe.
- Receives the deposit back.
- Rates the Owner.

Outcome: João obtains a suitable suit without high costs, through a simple and efficient process.

Scenario 2 – Marta publishes new items to the catalog
Actor: Marta (Owner)
Motivation: Monetize formal items that are rarely used.

Action:

- Logs into the Owner dashboard.
- Uploads photos of the item.
- Adds details such as size, color, style, and occasion.
- Sets the price and availability.
- Publishes the item to the catalog.

Outcome: The item becomes immediately available for rental.

Scenario 3 – Pedro manages rental requests
Actor: Pedro (Owner)
Motivation: Ensure that renters act responsibly.

Action:

- Receives a rental request.
- Reviews the renter's ratings and reputation.
- Approves the request.
- Arranges delivery with the shipping provider.
- Confirms the item's condition after return.

Outcome: The rental is completed safely and without damage.

Scenario 4 – Admin monitors platform activity
Actor: Admin
Motivation: Keep the platform free from abuse or fraud.

Action:

- Reviews reports and signs of anomalous behavior.
- Takes action when necessary (warnings, suspensions, bans).

Outcome: The platform remains stable, secure, and trustworthy.

2.3 Project epics and priorities

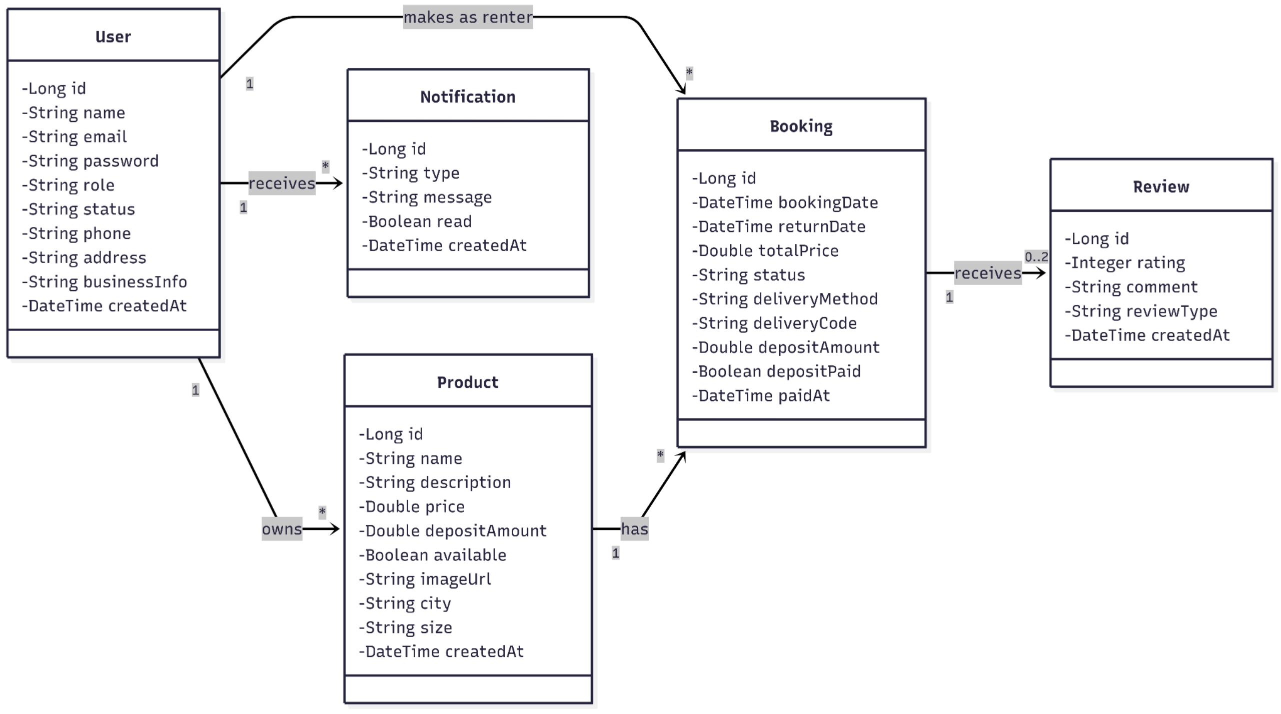
The main epics are:

- **Catalog and Search** - Owners create/manage products (with optional images, deposits, sizes, address/geo); geocode/reverse-geocode; renters browse available items with name/price filters; owners see their own available items; owner delete marks unavailable, admin delete removes with booking/notification cleanup; images served from uploads.
- **User and Role Management** - Sign-up/login with a unique email and BCrypt password; profile fields (phone, address, business info); roles renter/ owner/admin; admin can change roles (not to admin), suspend/ban/reactivate/ delete users with cascading booking/product cleanup; reputation endpoint summarizing ratings.
- **Reservations and Scheduling** - Bookings validate dates/conflicts; statuses PENDING_APPROVAL→APPROVED/REJECTED→PAID→COMPLETED/CANCELLED; approval requires delivery method (PICKUP with location, SHIPPING generates delivery code after payment); renter can cancel before start; owner sees pending queue; renter history/active views; product bookings viewable when paid/completed; notifications on key transitions.
- **Owner Dashboard** - “My bookings” for owners: list bookings on owned products (all and pending approval), plus notifications that inform booking/ payment/deposit events.
- **Renter Dashboard** - “My bookings” for renters: active bookings, history, payments/deposits, and related notifications.
- **Payments and Deposits** - Stripe Payment for approved bookings

(EUR, metadata booking/user/product); status check; marking paid update booking and delivery code for shipping; deposit flow: owner requests after return for paid/completed bookings (amount/reason), renter pays, owner can refund; deposit state stored per booking; no Stripe webhooks/settlement logic.

- **Reviews and Reputation System** - After COMPLETED, renter reviews the owner/ product (type OWNER) and owner reviews renter (type RENTER), one per type per booking; list reviews by product; user reputation aggregates averages/ counts per role and overall.
- **Admin** - Admin guard via X-User-Id + role; metrics (users by role, products, bookings by status, revenue/avg booking value); manage users (status/role, delete), manage products (list/delete with notifications), auto-cancel bookings when suspending/banning owners or renters; notifications for admin actions.

3 Domain model



User

The **User** entity represents any person registered on the platform. A user can assume different roles depending on their interactions with the system.

Characteristics:

- Users can act as **Renters**, **Owners**, or **Administrators**
- Each user has authentication credentials and profile information
- A reputation score is calculated based on reviews received

Responsibilities:

- Rent items as a renter
- List and manage items as an owner
- Manage platform operations as an administrator

Product

The **Product** entity represents a rental item listed on the platform by an owner.

Characteristics:

- Each product belongs to a single owner (User)
- Contains descriptive information (name, description)
- Includes pricing details and optional security deposit
- Stores location and availability information
- Supports image uploads

Responsibilities:

- Be discoverable through search and filtering
- Accept bookings based on availability
- Maintain historical booking records

Booking

The **Booking** entity represents a rental reservation created by a renter for a specific product.

Characteristics:

- Links one renter (User) to one product
- Tracks rental start and end dates
- Maintains approval, payment, and delivery states
- Manages deposit and refund information

Lifecycle:

- Pending approval
- Approved or rejected
- Paid
- Completed or cancelled

Review

The **Review** entity represents feedback provided after a completed booking.

Characteristics:

- Can be submitted by renters or owners
- Includes a numerical rating and optional comment
- Is linked to a specific booking
- Contributes to user reputation scores

Types of Reviews:

- Renter → Owner / Product
- Owner → Renter

Notification

The **Notification** entity represents system-generated messages sent to users.

Characteristics:

- Triggered by key events (booking updates, payments, disputes)
- Contains message content and timestamp
- Tracks read and unread status

Purpose:

- Inform users of important actions and state changes
- Improve transparency and user engagement

Concept Relationships

The relationships between the core domain entities are defined as follows:

1. A **User** can own multiple **Products** (Owner role)
2. A **User** can create multiple **Bookings** (Renter role)
3. A **Product** can be associated with multiple **Bookings** over time

4. A **Booking** links one **Renter** (User) with one **Product** and its **Owner**
5. A **Booking** can have up to two **Reviews** (one from the renter and one from the owner)
6. A **User** can receive multiple **Notifications**

4 Architecture notebook

4.1 Key requirements and constrains

The architecture of our system results from an in-depth analysis of both functional requirements and broader architectural quality attributes. These requirements guide the selection of technologies and the organization of system components to ensure **performance, maintainability, scalability, and adaptability**, within the scope and constraints of this course.

This section identifies the key **requirements, constraints, and design drivers** that shape the proposed system architecture.

4.1.1 Functional Requirements

Functional requirements define the core capabilities that the system must provide to end users in order to fulfill the business and operational goals established in the case study.

1. **Multi-role user management**
   The system must support different user roles (Renters, Owners, and Administrators), each with distinct permissions and available functionality.
2. **Item catalog search and discovery**
   Users must be able to browse and search available rental items using filters such as name, price range, and municipality location.
3. **Booking and availability management**
   The system must prevent overlapping bookings and double-booking scenarios, ensuring real-time availability updates and owner approval workflows.
4. **Approval and logistics flow**
   Owners must be able to approve or reject booking requests, select delivery methods (pickup or shipping), and trigger delivery code generation when applicable.
5. **Payment and deposit support**
   Integration with the Stripe payment gateway must support rental payments and optional security deposits, including refund and retention workflows.
6. **Owner tools (item management)**
   Owners must be able to create, update, and delete listings, upload product images, and

configure pricing and deposit requirements.

7. **User dashboard and history**
 Users must have access to dashboards displaying booking history, spending or earnings statistics, and notifications in a clear and intuitive interface.

8. **Reviews and reputation system**
 The platform must support a bidirectional review system, allowing renters and owners to rate each other after completed bookings.

9. **Administrative backoffice**
 Administrators must be able to monitor platform metrics, manage users (suspend, ban, delete), and moderate product listings.

4.1.2 Non-Functional Requirements

Non-functional requirements define the quality attributes that ensure the system is robust, scalable, and production-ready. These characteristics guide architectural decisions and support long-term sustainability.

1. **Scalability**
 The system must support growth in users, products, and bookings, handling concurrent read/write operations with consistent performance.

2. **Performance and responsiveness**
 Real-time operations such as booking updates and notifications must remain responsive, even under high load.

3. **Platform support (User Interface)**
 The frontend must be responsive and accessible across desktop and mobile web platforms, ensuring a consistent user experience.

4. **Integration with external systems**
 External integrations (Stripe payments, municipality validation APIs, image storage) must be loosely coupled and abstracted for testability.

5. **Geolocation support**
 Municipality-based filtering must be supported, including Portuguese municipality validation and autocomplete functionality (by GeoApi).

6. **Fault tolerance and resilience**
 The system must gracefully handle failures in external services, particularly payment and third-party APIs.

7. **Security**
 Role-based authorization, secure password hashing, protected endpoints, and strict permission validation must be enforced.

8. **Maintainability and testability**
   The architecture must support modular design, automated testing (unit, integration, end-to-end), and CI/CD pipelines.

9. **Observability**
   Logging, health checks, and monitoring mechanisms must be implemented to support debugging and performance tracking.

10. **Portability**
    The system must be containerized using Docker and Docker Compose to ensure consistent deployments across environments.

4.1.3 Architectural Design Drivers

The following architectural drivers have significantly influenced the system design and technology choices:

Driver	**Architectural Decision**
Need for responsive web and mobile UI	Use of React with TypeScript and Tailwind CSS for responsive frontend development
Integration with payment provider	Stripe API integration through an abstracted payment service layer
Multi-role authorization requirements	Role-based access control with user status management (active, suspended, banned)
Scalability and extensibility	Layered monolithic architecture with clear separation of controllers, services, and repositories
User dashboards and history tracking	Stateful backend services managing bookings, notifications, and reputation data
Real-time availability enforcement	Database-level validation to prevent overlapping bookings
Image upload and storage	File-based image storage using UUID naming and dedicated serving endpoints
Location-based filtering	Integration with Portuguese municipality APIs for validated autocomplete
Development speed with maintainability	CI/CD pipelines using GitHub Actions, SonarCloud analysis, and Docker-based deployment
Comprehensive testing strategy	Multi-layer testing with JUnit (unit), Cucumber/Playwright (E2E), and k6 (load testing)

4.2 Architecture view

The architecture for BlackTie follows a layered architectural style, organising the software into distinct functional blocks that interact through well-defined interfaces. The main building blocks are grouped into packages or modules according to their business responsibilities. The main logical packages are:

- User Management (handling registration, authentication, profile management, and role switching between Renter and Owner)
- Catalog Management (managing product listings, search, filtering by name, price, and municipality)
- Booking Management (handling booking creation, approval workflows, cancellations, and delivery logistics)
- Review Management (enabling bidirectional reviews between renters and owners after completed rentals)
- Payment Integration (coordinating rental payments and security deposits with an external service, Stripe)
- Notification Management (delivering system notifications for booking updates, payments, and platform actions)
- Admin Management (providing administrative controls, user moderation, and platform metrics)

All communication between the modules is typically via standard HTTP protocols, with REST conventions applied for API endpoints. The core system follows separation of concerns, where the API layer never contains business logic and the business logic never directly interacts with the database. Repositories are responsible for that.

Interactions Between Modules:

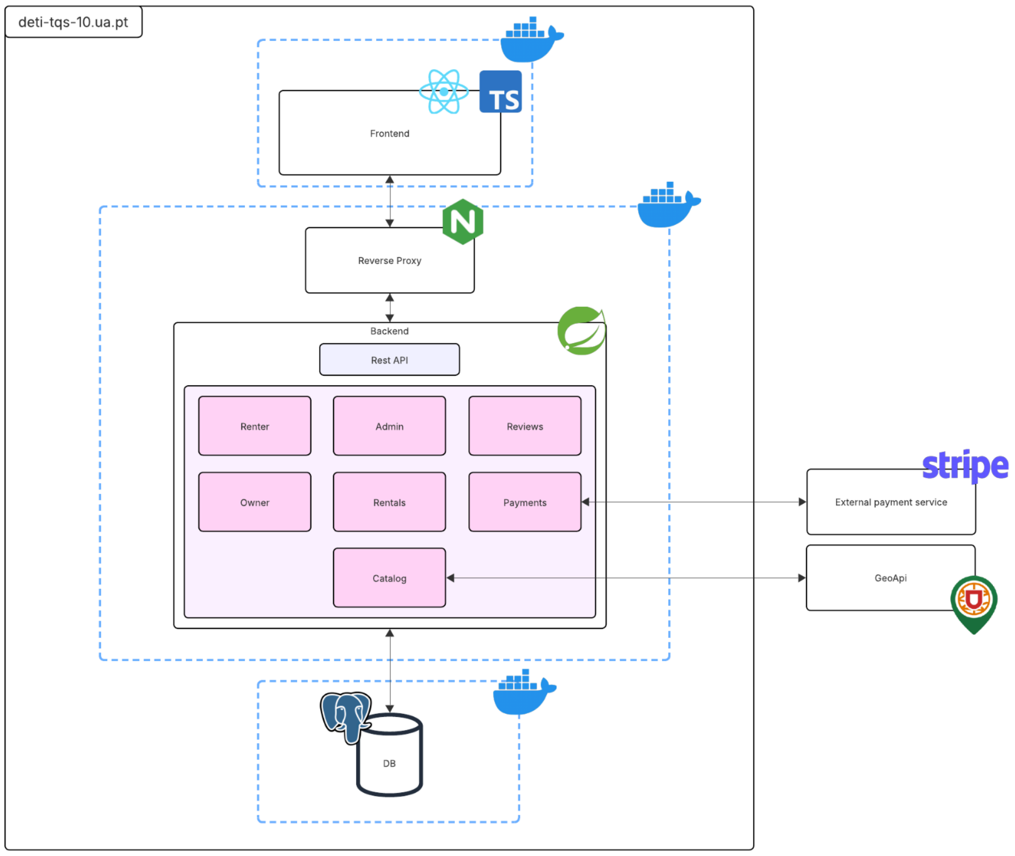
For example, when a user makes a booking:

- The API receives the booking request and forwards it to the Booking Service.
- The Booking Service validates availability, calculates the total cost (rental price plus optional deposit), and updates the relevant Product and Booking entities.
- Since payment is required, the Booking Service communicates with the Payment Integration module, which in turn talks to the external payment provider (Stripe).
- Upon successful payment, the Notification Service sends confirmation notifications to both the renter and the owner.
- All changes are persisted through the appropriate repositories.

Advanced Design Considerations:

- External Service Integration: The system integrates with third-party services (Stripe for payment processing, GeoAPI for Portuguese municipality validation), using secure HTTPS protocols and standardized APIs.
- Data Synchronization: Consistency is maintained using transactional updates within the service layer. For real-time availability, the architecture supports near real-time updates to prevent double bookings, but global distributed synchronization is not required for this MVP.
- Distributed Workflows: While the main application is monolithic at this stage, the logical separation allows for future migration to a microservices architecture, where services like payments or notifications could run as independent modules.

4.3 Deployment view (production configuration)



This section presents a high-level overview of our system's deployment and architecture. It describes how the main components interact to deliver a web-based solution for browsing, renting, and managing formal wear items.

Production Domain: `deti-tqs-10.ua.pt`

System Architecture Overview

Frontend

The frontend is a web application built with **React** and **Vite**. It provides the user interface for browsing products, managing bookings, and performing payments. Client-side routing is handled with React Router, and the Stripe SDK is used to support secure checkout flows.

Reverse Proxy

An **Nginx** reverse proxy is used to serve the frontend application and route incoming requests. Static content is served directly, while API and documentation requests are forwarded to the backend service. Security-related HTTP headers are configured to mitigate common web vulnerabilities.

Backend

The backend is implemented using **Spring Boot** and **Java 21**. It follows a modular architecture with separate domains for catalog management, rentals, payments, users, administration, and reviews. Each module follows a layered structure separating controllers, services, and data access logic.

The backend exposes a REST API documented using Swagger UI.

Database

A **PostgreSQL** database is used for persistent storage of all application data, including users, products, bookings, and transactions. Data access is handled through Spring Data JPA.

External Services

The platform integrates with external services to support core functionality:

- **Stripe** is used for secure payment processing
  - **GeoAPI** is used for address validation and geocoding within Portugal

Deployment Configuration

The application is deployed using **Docker Compose**, with each major component running in its own container.

Service	**Purpose**
Frontend	Web interface and reverse proxy
Backend	REST API and business logic
Database	Persistent data storage

All services communicate over a private Docker network, while only the frontend is exposed publicly.

CI/CD Pipeline

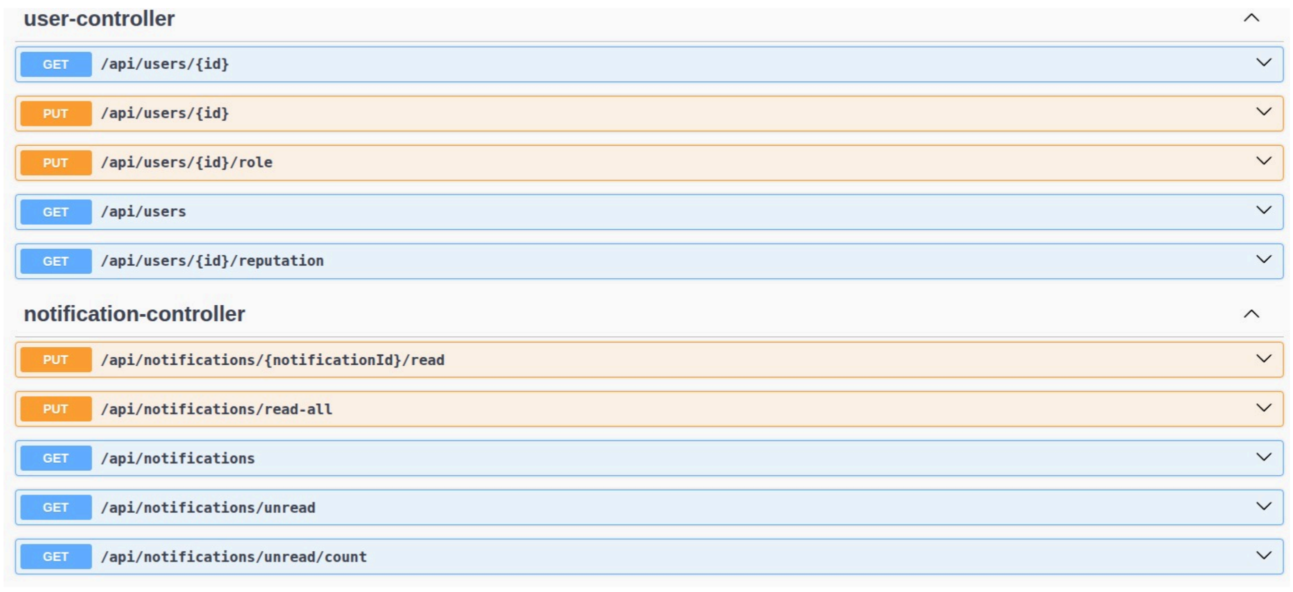
Continuous integration is implemented using **GitHub Actions**. The pipeline is triggered on pushes and pull requests to the development branch and includes:

- Backend build and unit testing
- Frontend build and validation
- Code quality analysis with SonarCloud

Security Considerations

- Sensitive configuration is provided through environment variables
- Communication with external services uses secure HTTPS connections
- HTTP security headers are enforced at the reverse proxy level

5 API for developers

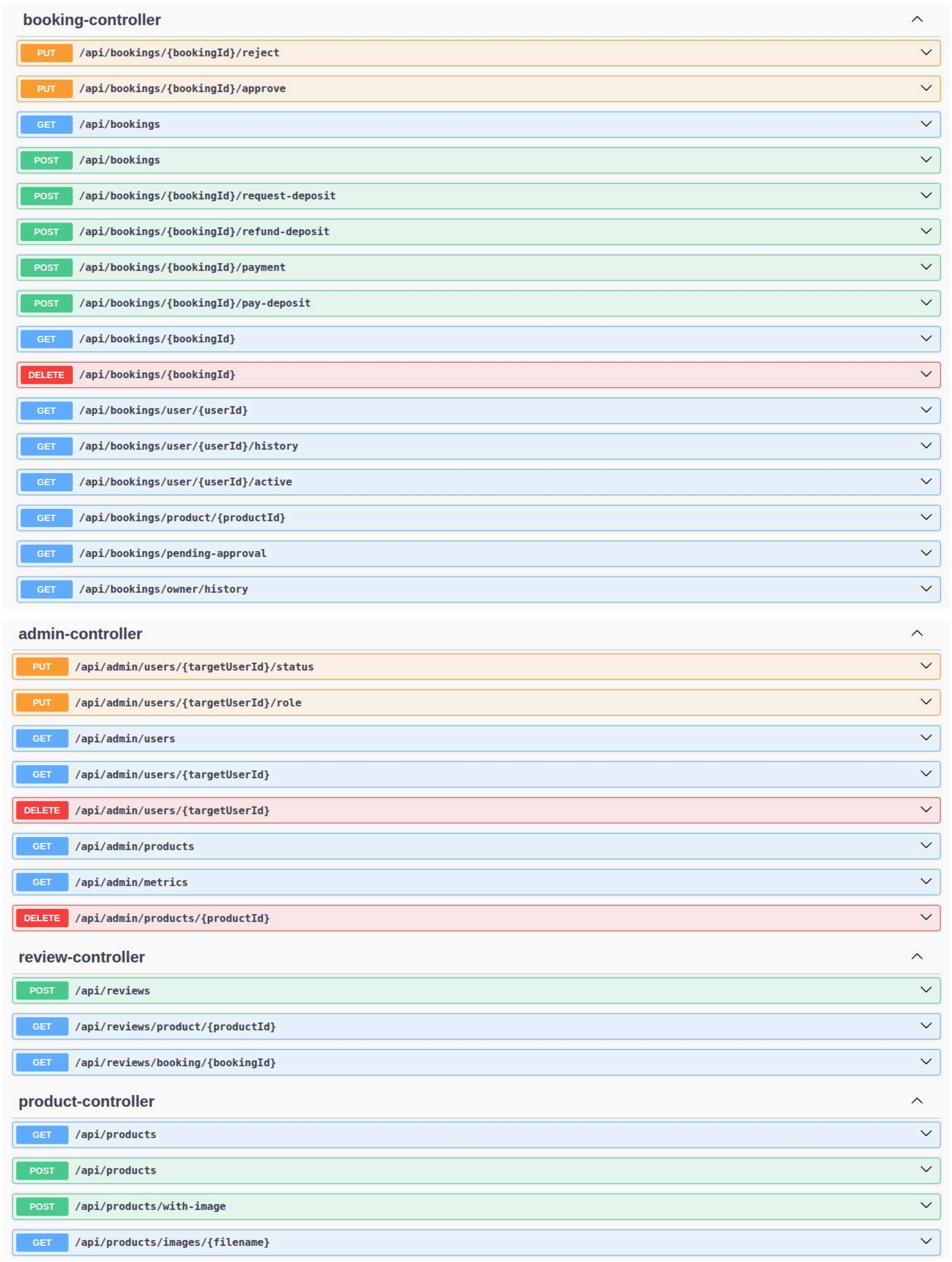


user-controller

```
GET /api/users/{id}
PUT /api/users/{id}
PUT /api/users/{id}/role
GET /api/users
GET /api/users/{id}/reputation
```

notification-controller

```
PUT /api/notifications/{notificationId}/read
PUT /api/notifications/read-all
GET /api/notifications
GET /api/notifications/unread
GET /api/notifications/unread/count
```



booking-controller

```
PUT     /api/bookings/{bookingId}/reject
PUT     /api/bookings/{bookingId}/approve
GET     /api/bookings
POST    /api/bookings
POST    /api/bookings/{bookingId}/request-deposit
POST    /api/bookings/{bookingId}/refund-deposit
POST    /api/bookings/{bookingId}/payment
POST    /api/bookings/{bookingId}/pay-deposit
GET     /api/bookings/{bookingId}
DELETE  /api/bookings/{bookingId}
GET     /api/bookings/user/{userId}
GET     /api/bookings/user/{userId}/history
GET     /api/bookings/user/{userId}/active
GET     /api/bookings/product/{productId}
GET     /api/bookings/pending-approval
GET     /api/bookings/owner/history
```

admin-controller

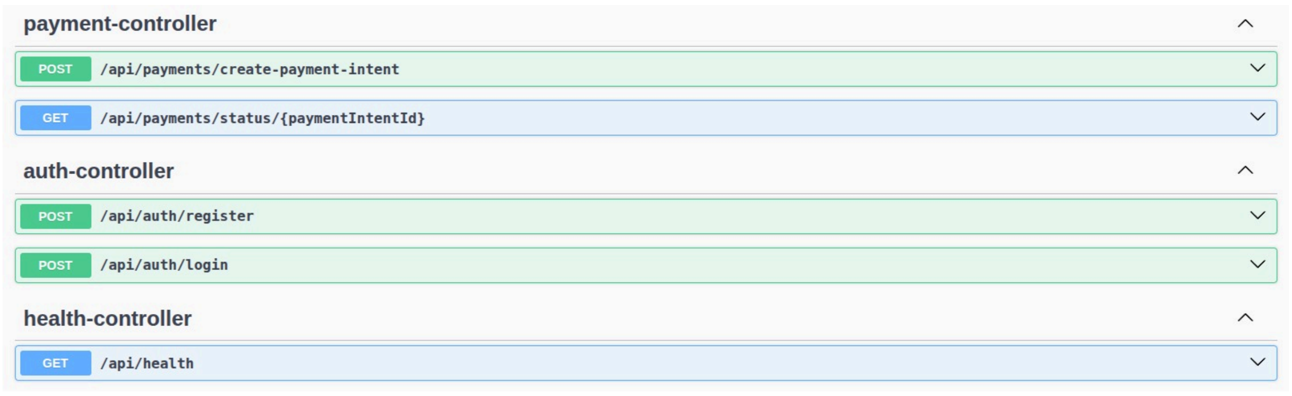
```
PUT     /api/admin/users/{targetUserId}/status
PUT     /api/admin/users/{targetUserId}/role
GET     /api/admin/users
GET     /api/admin/users/{targetUserId}
DELETE  /api/admin/users/{targetUserId}
GET     /api/admin/products
GET     /api/admin/metrics
DELETE  /api/admin/products/{productId}
```

review-controller

```
POST    /api/reviews
GET     /api/reviews/product/{productId}
GET     /api/reviews/booking/{bookingId}
```

product-controller

```
GET     /api/products
POST    /api/products
POST    /api/products/with-image
GET     /api/products/images/{filename}
```


payment-controller

POST /api/payments/create-payment-intent

GET /api/payments/status/{paymentIntentId}

auth-controller

POST /api/auth/register

POST /api/auth/login

health-controller

GET /api/health

The backend exposes a **RESTful API**, designed according to industry best practices for **resource-oriented architectures**.

The API is organized around the platform's core business entities, each managed by a dedicated controller.

All endpoints are fully documented and can be **explored and tested interactively** using **Swagger UI**, available at:

`http://deti-tqs-10.ua.pt/docs`

API Overview

The API provides functionality for user authentication, product management, bookings, payments, reviews, notifications, and platform administration.
Each major feature is exposed through a dedicated API collection.

Main API Collections

Authentication (`/api/auth`)

Handles user registration and authentication.

Responsibilities:

- User signup with email, name, and password
- User login and authentication

Description:
New users can create accounts, while existing users authenticate to access their bookings, listings, and profile information.

Users (`/api/users`)

Manages user profiles and roles.

Responsibilities:

- Retrieve user profiles
- Update profile information
- Manage user roles (Renter / Owner)
- View user reputation and ratings

Description:
Users can be retrieved by ID, update their personal information, and view reputation metrics derived from reviews.

Products (`/api/products`)

Manages rental item listings.

Responsibilities:

- Create product listings (with or without images)
- Upload and serve product images
- Retrieve available products with optional filters
- Delete product listings

Filters Supported:

- Name
- Maximum price

Description:
Product creation supports multipart requests for image uploads. Listings can be searched, filtered, and removed by owners or administrators.

Bookings (`/api/bookings`)

Manages the full booking lifecycle.

Responsibilities:

- Create booking requests
- View bookings by renter, owner, or product

- Approve or reject booking requests
- Cancel bookings
- Manage deposits (request, pay, refund)
- Retrieve booking history and active bookings

Description:
These endpoints coordinate the interaction between renters and owners, handling approvals, logistics, and booking state transitions.

Reviews (`/api/reviews`)

Handles review creation and retrieval.

Responsibilities:

- Submit reviews after completed bookings
- Retrieve reviews by booking or product

Description:
Reviews contribute to user reputation and help build trust across the platform. Reviews include ratings and optional comments.

Payments (`/api/payments`)

Handles payment processing via **Stripe**.

Responsibilities:

- Create payment intents
- Verify payment status

Key Endpoints:

- `/api/payments/create-payment-intent`
- `/api/payments/status/{paymentIntentId}`

Description:
All payments are securely processed through Stripe, supporting both rental payments and security deposits.

Notifications (`/api/notifications`)

Manages system notifications for users.

Responsibilities:

- Retrieve all notifications
- Retrieve unread notifications
- Get unread notification count
- Mark notifications as read (single or bulk)

Description:
Notifications inform users about booking updates, payments, disputes, and platform actions.

Admin (`/api/admin`)

Provides administrative control over the platform.

Responsibilities:

- View platform-wide metrics
- Manage users (view, update status, update role, delete)
- Manage products (view, delete)

Access Control:

- All admin endpoints require **administrator privileges**

Description:
These endpoints allow administrators to monitor platform health, enforce policies, and manage users and content.

Health (`/api/health`)

Provides backend availability checks.

Responsibilities:

- Return current system health status

Description:
Used for monitoring, deployment checks, and system diagnostics.

Resource-Oriented Design Principles

The BlackTie API follows RESTful conventions:

- Resource names use **plural nouns** (e.g., `/users`, `/products`, `/bookings`)
- HTTP methods map directly to actions:
    - `GET` – Retrieve resources
    - `POST` – Create resources
    - `PUT` / `PATCH` – Update resources
    - `DELETE` – Remove resources
- Resource identifiers are passed as **path parameters** (e.g., `/api/bookings/{bookingId}`)

API Documentation and Exploration

The **complete and always up-to-date API documentation** is provided through **Swagger UI**, accessible at:

`/docs`

Swagger UI allows:

- Browsing all available endpoints
- Inspecting request and response schemas (DTOs)
- Testing API calls directly from the browser

This ensures both developers and stakeholders have a clear, interactive view of the system's capabilities.