# Final Project Report

## Web Services Programming – Eiffel Bike Corp

Marcos Duchinski
vianaduchinski@edu.univ-eiffel.fr
Pedro Moura
pedro.ferra-moura@edu.univ-eiffel.fr
Max Aguirre
max.aguirre@edu.univ-eiffel.fr

# Table of contents

---

## Document control

**Document title:** *Eiffel Bike Corp — Backend API Report* **Project:** Eiffel Bike Corp (bike rental + corporate resale API)
**Scope:** Backend API **Format:** Markdown

### Status

- **Status:** Draft
- **Team:** Marcos / Max / Pedro
- **Last updated:** 2025-12-26

### Versioning

| Version | Date (YYYY-MM-DD) | Author(s) | Changes |
|---|---|---|---|
| 0.1 | 2025-12-26 | Team | Initial structure + backend overview |

### Audience

- Course instructors / reviewers
- Developers (backend)
- Testers / integrators (API consumers)

### Useful URLs (local dev)

- **API base URL:** `http://localhost:8080/api`

- **Swagger UI:** `http://localhost:8080/swagger-ui/index.html`

---

# 1. Project description

## 1.1 Context

Eiffel Bike Corp provides a **bike rental service inside Université Gustave Eiffel**, allowing **students and employees** to offer bikes for rent, and allowing customers (university members) to **search, rent, pay, and return** bikes. When a requested bike is unavailable, customers are placed on a **waiting list**, and they receive a **notification** when the bike becomes available.

In addition, the system supports a second business flow: EiffelBikeCorp can **sell corporate bikes** under controlled conditions (e.g., only bikes that have been rented at least once), enabling customers to browse offers, place items in a basket, check out, and pay.

## 1.2 Objectives

The backend API aims to:

- Expose a clear set of **REST endpoints** that map to the user stories.
- Provide a consistent **domain model** for rental + sale workflows.
- Ensure **fairness and correctness** in key flows:
    - waiting list (FIFO / first-come, first-served),
    - availability and state transitions (bike/rental/sale offer),
    - money consistency with **EUR as the system currency**.
- Provide production-like integrations:
    - **Stripe** for payment processing,
    - **ExchangeRate-API** for multi-currency conversion.
- Offer a complete, usable **API contract** via Swagger/OpenAPI.

## 1.3 Functional scope

Included:

- User registration and login (JWT-based authentication)
- Bike catalog and bike offering for rent
- Search rentable bikes
- Rent a bike or join the waiting list
- Notifications when bikes become available
- Pay rentals with currency conversion
- Return bikes with return notes
- Corporate resale: sale offers, sale notes, search offers, offer details/availability
- Basket management, checkout, purchase payment
- History endpoints (purchase history; rental history if enabled)

Not included (at this stage):

- Advanced admin/backoffice features

- Refund workflows and complex payment dispute handling
- Full observability stack (tracing, metrics dashboards), beyond basic logging

### 1.4 Non-functional goals and constraints

- **Technology constraint:** Java RS approach using **JAX-RS (Jersey)** hosted in **Spring Boot**
- **Persistence:** PostgreSQL with JPA/Hibernate mappings
- **Documentation:** Swagger/OpenAPI accessible via Swagger UI
- **Deployment support:** Docker for local reproducibility

### 1.5 Technology overview (backend)

- **Runtime:** Java (Spring Boot)
- **HTTP framework:** Jersey / JAX-RS
- **Persistence:** Spring Data JPA + Hibernate + PostgreSQL
- **API documentation:** Swagger OpenAPI (swagger-jaxrs2-jakarta + annotations)
- **Payments:** Stripe (`https://stripe.com/docs/api`)
- **FX conversion:** ExchangeRate-API (`https://www.exchangerate-api.com/docs/overview`)
- **Containerization:** Docker + Docker Compose

---

## 2. Scope, assumptions, and glossary

### 2.1 Scope

This section focuses **only on the backend API** of the Eiffel Bike Corp system.

Included in scope:

- REST endpoints implemented in the backend (JAX-RS/Jersey hosted in Spring Boot)
- DTO contracts (requests/responses) and status codes
- Domain model and business rules that enforce the user stories
- Authentication and authorization model (JWT bearer)
- Integrations:
    - PostgreSQL persistence
    - Docker Compose runtime (DB + optional app)
    - Swagger/OpenAPI documentation
    - Stripe payment processing
    - ExchangeRate-API currency conversion

Out of scope:

- Production deployment hardening (Kubernetes, CI/CD pipelines, secrets manager)
- Full monitoring/observability platform (metrics dashboards, distributed tracing)
- Refund and dispute management workflows (beyond basic payment status transitions)
- Performance benchmarking and load testing (can be added later)

---

### 2.2 Assumptions

The backend design and examples in this report assume:

1. **Environment assumptions**

- Local development uses:
  - `http://localhost:8080` for backend
  - PostgreSQL running via Docker on `localhost:5432`
- Swagger UI is reachable at:
  - `http://localhost:8080/swagger-ui/index.html`

2. **Security assumptions**

- Authentication is **JWT-based**.
- Secured endpoints expect:
  - `Authorization: Bearer <accessToken>`
- The token contains the user identifier (UUID) used to resolve the authenticated customer/provider in secured use cases.

3. **Domain/business assumptions**

- Bikes have a lifecycle (e.g., `AVAILABLE`, `RENTED`, `MAINTENANCE`).
- Rentals have a lifecycle (e.g., `ACTIVE`, `CLOSED`, `CANCELED`).
- Sale offers have a lifecycle (e.g., `LISTED`, `SOLD`, `REMOVED`).
- Waiting lists follow **FIFO** ("first come, first served").
- EUR is the internal "system currency"; multi-currency payments are converted to EUR and stored with an FX snapshot.

4. **Integration assumptions**

- Stripe is reachable and configured through the application configuration.
- ExchangeRate-API calls succeed under normal conditions; a short cache may be used to reduce repeated calls.

---

## 2.3 Glossary

| Term | Meaning |
|---|---|
| API | Application Programming Interface; here, the REST endpoints exposed by the backend |
| JAX-RS / Java RS | Java API for RESTful Web Services (Jakarta REST) used to implement controllers/resources |
| Jersey | A reference implementation of JAX-RS used to run JAX-RS resources inside Spring Boot |
| DTO | Data Transfer Object; request/response models used at the API boundary |
| Provider | Entity that offers bikes for rent (Student, Employee, EiffelBikeCorp) |
| Customer | Entity that rents bikes and/or buys bikes |
| Waiting list | Queue for a bike when it is unavailable; serves customers in FIFO order |
| Notification | Message emitted when a bike becomes available for a waiting customer |

| Term | Meaning |
|---|---|
| Basket | Temporary container of sale offers selected by a customer before checkout |
| Checkout | Operation that converts a basket into a Purchase |
| Purchase | A record of items bought by a customer (result of checkout) |
| Payment gateway | External provider processing card payments (Stripe) |
| FX / Exchange rate | Conversion rate between currencies used to convert payments into EUR |
| FX snapshot | Persisted exchange rate value used at payment time for audit and consistency |
| Swagger UI | Web UI that visualizes the OpenAPI contract and allows interactive API calls |
| OpenAPI | Specification for describing REST APIs (operations, schemas, auth, etc.) |
| JWT | JSON Web Token used for stateless authentication |

# 3. Requirements mapping (User Stories → API → Domain Model)

This section links the functional requirements (user stories) to the REST API endpoints and the main domain objects that implement them. It also defines the main "happy-path" usage scenarios that will be referenced later in the user manual.

## 3.1 User stories (grouped)

The project requirements are expressed as user stories covering two phases: **rental inside the university** and **sale to the outside world**, with payments and currency conversion.

**A) Offering bikes for rent (Providers)**

- **US_01:** As a Student, I want to offer my bike for rent so that other university members can rent it.
- **US_02:** As an Employee, I want to offer my bike for rent so that other university members can rent it.
- **US_03:** As EiffelBikeCorp, I want to offer company bikes for rent so that customers can rent bikes even when no private bikes are available.

**B) Searching, renting, waiting list, notifications, paying, returning (Customers)**

- **US_04:** As a Customer, I want to search for bikes available for rent so that I can quickly find a suitable bike.
- **US_05:** As a Customer, I want to rent a bike so that I can use it for my commute or daily needs.
- **US_06:** As a Customer, I want to be added to a waiting list when a bike is unavailable so that I can rent it as soon as it becomes available.
- **US_07:** As a Customer, I want to receive a notification when a bike becomes available so that I can rent it before someone else does.
- **US_08:** As a Customer, I want to pay the rental fee in any currency and have it converted to euros so that I can pay in my preferred currency while the system remains consistent.
- **US_09:** As a Customer, I want to add notes when returning a bike so that the next renter and the bike provider know the bike's condition.

**C) Offering for sale (EiffelBikeCorp)**

- **US_10:** As EiffelBikeCorp, I want to list for sale only company bikes that have been rented at least once so that only used corporate bikes are eligible for resale.

- **US_11:** As EiffelBikeCorp, I want to offer bikes for sale with detailed notes so that buyers can assess each bike's condition before purchasing.

**D) Shopping & paying (Customers)**

- **US_12:** As a Customer, I want to search for bikes available to buy so that I can find a bike to purchase.
- **US_13:** As a Customer, I want to view the sale price of bikes offered for sale so that I can compare options before buying.
- **US_14:** As a Customer, I want to view the notes associated with bikes offered for sale so that I can make an informed buying decision.
- **US_15:** As a Customer, I want to check whether a bike offered for sale is still available so that I don't try to buy a sold or unavailable bike.
- **US_16:** As a Customer, I want to add bikes offered for sale to a basket so that I can prepare my purchase before paying.
- **US_17:** As a Customer, I want to remove bikes from my basket so that I can adjust my selection.
- **US_18:** As a Customer, I want to purchase the bikes in my basket so that I can complete the transaction.
- **US_19:** As a Customer, I want to pay for my purchase through a payment gateway so that the system can verify funds and complete the payment.

**E) History**

- **US_20:** As a Customer, I want to view my purchase history so that I can track what I bought and when.
- **US_21:** As a Customer, I want to view my rental history so that I can track what I rent and when.

---

## 3.2 Traceability matrix (US → REST API → Main domain objects)

| User Story | Main Endpoint(s) | Key domain objects (conceptual model) |
|---|---|---|
| US_01 / US_02 / US_03 Offer bike for rent | `POST /api/bikes` | `Bike` (offeredBy, rentalDailyRateEur, status) + `BikeProvider` (Student/Employee/Corp) |
| US_04 Search bikes to rent | `GET /api/bikes?status=...&q=...&offeredById=...` | `Bike` (+ filters on status/description/provider) |
| US_05 Rent a bike / US_06 Waitlist if unavailable | `POST /api/rentals` (201 RENTED / 202 WAITLISTED) | `Rental` + `WaitingList` + `WaitingListEntry` |
| US_06 View waitlist | `GET /api/rentals/waitlist` | `WaitingListEntry` (+ customer + servedAt) |
| US_07 Notifications | `GET /api/rentals/notifications` | `WaitingListEntry.notifications` (notifications attached to waitlist entries) |
| US_08 Pay rental (any currency → EUR) | `POST /api/payments/rentals` | `Rental` + `RentalPayment` (recorded per rental) |
| US_09 Return bike + notes | `POST /api/rentals/{rentalId}/return` and `GET /api/bikes/{bikeId}/return-notes` | `Rental.returnNote` (1 per rental) |

| User Story | Main Endpoint(s) | Key domain objects (conceptual model) |
|---|---|---|
| US_10 Create sale offer (corp + rented at least once) | `POST /api/sale-offers` | `SaleOffer` (seller=EiffelBikeCorp, bike, status) + uniqueness per bike |
| US_11 Add detailed sale notes | `POST /api/sale-offers/notes` | `SaleOffer.notes` |
| US_12–US_15 Search + view details/availability/price/notes | `GET /api/sale-offers?q=...` and `GET /api/sale-offers/{saleOfferId}` | `SaleOffer` (askingPriceEur, status, notes, buyer) |
| US_16 Add to basket | `GET /api/basket` and `POST /api/basket/items` | `Basket` + `BasketItem(offer, priceSnapshot)` |
| US_17 Remove from basket | `DELETE /api/basket/items/{saleOfferId}` | `BasketItem` unique per (basket, offer) |
| US_18 Checkout | `POST /api/purchases/checkout` | `Purchase` + `PurchaseItem` created from basket |
| US_19 Pay purchase | `POST /api/payments/purchases` | `SalePayment` linked to `Purchase` (amountEur + fxRate + gateway id) |
| US_20 Purchase history | `GET /api/purchases` | `Purchase` list by customer |
| US_21 Rental history | `GET /api/rentals` | `Rental` list by customer |

### 3.3 Conceptual coverage and business-rule anchors in the domain model

A few domain modeling choices directly enforce requirements:

- **One waiting list per bike:** `Bike` owns a one-to-one `WaitingList`, and `WaitingList` enforces uniqueness on `bike_id`.
- **First-come, first-served waiting list:** `WaitingList.entries` are ordered by `createdAt ASC`, matching FIFO behavior at the persistence level.
- **One sale offer per bike:** `Bike.saleOffer` exists and `SaleOffer` has a unique constraint on `bike_id`.
- **Basket and purchase snapshots:** basket/purchase items store a `unitPriceEurSnapshot`, preventing price drift between adding to basket and checkout.
- **Payments store FX + EUR for consistency:** `SalePayment` stores `originalCurrency`, `fxRateToEur`, and `amountEur` (auditable conversion).

---

### 3.4 Happy-path scenarios of use (backend perspective)

These scenarios are the baseline flows used later in the user manual and test strategy.

**Scenario S1 — Offer a bike for rent (US_01/02/03)**

1. Provider calls `POST /api/bikes` with description + daily rate + provider identity.
2. System persists `Bike` with `status=AVAILABLE` and links it to the provider (`Bike.offeredBy`).

**Scenario S2 — Search and rent a bike (US_04 + US_05)**

1. Customer searches: `GET /api/bikes?status=AVAILABLE&q=...`

2. Customer rents: `POST /api/rentals` → response `201` with `result=RENTED`.

**Scenario S3 — Waitlist and notification (US_06 + US_07)**

1. Customer attempts to rent an unavailable bike: `POST /api/rentals` → response `202` with `result=WAITLISTED`.
2. Customer can consult the waitlist: `GET /api/rentals/waitlist`.
3. When a bike becomes available, notifications become visible via `GET /api/rentals/notifications`.

**Scenario S4 — Pay and return with notes (US_08 + US_09)**

1. Pay rental: `POST /api/payments/rentals` (any currency; stored consistently in EUR).
2. Return bike with notes: `POST /api/rentals/{rentalId}/return`.
3. View bike return-notes history: `GET /api/bikes/{bikeId}/return-notes`.

**Scenario S5 — Offer for sale, shop, check out, and pay (US_10–US_19)**

1. Corp lists bike: `POST /api/sale-offers` (rule documented in endpoint description).
2. Corp adds notes: `POST /api/sale-offers/notes`.
3. Customer searches: `GET /api/sale-offers?q=...` and opens details: `GET /api/sale-offers/{id}`.
4. Customer adds/removes from basket: `POST /api/basket/items`, `DELETE /api/basket/items/{saleOfferId}`.
5. Check out: `POST /api/purchases/checkout`.
6. Pay purchase: `POST /api/payments/purchases`.

**Scenario S6 — History (US_20 + US_21)**

- Purchase history: `GET /api/purchases`.
- Rental history: `GET /api/rentals`.

---

# 4. Conceptual model

This section describes the main domain concepts and their relationships that implement the user stories. The model is designed to enforce key business rules and invariants while supporting the required workflows.

Class Diagram:  Class diagram

## 4.1 Domain overview

The backend is organized around two closely related subdomains:

1. **Rental subdomain** (university bike sharing)

- Providers (students, employees, EiffelBikeCorp) offer bikes for rent.
- Customers search bikes, rent them, and return them with notes.
- If a bike is not available, customers join a waiting list (FIFO) and receive notifications when the bike becomes available.

2. **Sale subdomain** (corporate resale)

- EiffelBikeCorp lists eligible corporate bikes for sale.
- Customers browse sale offers, view price and notes, add items to a basket, check out, and pay.
- Purchase history is available to customers.

Cross-cutting both subdomains:

- **Payments** (rental payments + purchase payments)
- **Currency conversion** using an external exchange-rate provider (store EUR + conversion snapshot)

The class diagram provides the conceptual entities and their relationships.

---

## 4.2 Core actors and identities

**BikeProvider (abstract)**

A provider is the "owner" offering bikes for rent. Providers are modeled via an abstract type with concrete specializations:

- **Student**
- **Employee**
- **EiffelBikeCorp**

This supports US_01–US_03 without duplicating the "offer bike" logic per role; the provider type becomes data and policy, not separate endpoint logic.

**Customer**

A **Customer** is the actor that rents bikes and buys bikes. Customers are linked to:

- Rentals
- Waiting list entries
- Purchases
- Basket

# This separation (Provider vs Customer) avoids ambiguity: a student/employee can act as both a provider (offering a bike) and a customer (renting/buying) through different aspects of their user profile, not duplicate accounts.

## 4.3 Rental subdomain

**Bike**

A **Bike** is the core object in the rental domain. It has:

- a **status** (availability lifecycle)
- a link to its **provider**
- a **daily rental price in EUR**

A bike can also be related to:

- **WaitingList** (for unavailable bikes)
- **SaleOffer** (if corporate and eligible for resale)

This single "Bike" entity supports both rental and resale flows while keeping the rental state machine explicit.

**Rental**

A **Rental** represents a time-bounded usage of a bike by a customer:

- links **Customer** ↔ **Bike**

- has timestamps (start/end)
- has a rental **status**
- has a computed **totalAmountEur**
- can have a single **ReturnNote**
- can have associated **RentalPayment** records

Rentals implement the core of US_05 (rent), US_09 (return + note), and support history queries (US_21).

**ReturnNote**

A **ReturnNote** captures condition information provided at return time:

- author (customer)
- comment/condition
- timestamp

Conceptually, it improves trust and bike quality transparency for future renters and providers (US_09).

**WaitingList, WaitingListEntry, Notification**

When a bike is unavailable, customers may join a **WaitingList** for that bike:

- Each bike has at most one waiting list.
- A waiting list contains ordered **WaitingListEntry** items.
- Notifications are linked to entries to record that a specific user was notified when availability changed.

This model supports:

- US_06 (waitlist join + later fulfillment)
- US_07 (notifications)
- fairness via FIFO ordering

---

## 4.4 Sale subdomain

**SaleOffer**

A **SaleOffer** represents a corporate bike offered for sale:

- links to exactly one **Bike**
- is owned/sold by **EiffelBikeCorp**
- has an asking price (EUR)
- has a sale lifecycle (listed/sold/removed)
- has zero or one buyer

`SaleOffer` is core for US_10 (eligibility to list) and enables US_12–US_15 (search/view/availability).

**SaleNote**

A **SaleNote** contains detailed condition notes attached to a sale offer (US_11, US_14). This lets buyers judge a bike before purchasing.

**Basket and BasketItem**

A **Basket** belongs to one customer and represents a "pre-checkout" selection of sale offers.

- A basket contains **BasketItem** entries, each referencing a sale offer.
- Basket items store a **unitPriceEurSnapshot** to prevent price drift between browsing and checkout.

This directly supports US_16 and US_17.

**Purchase and PurchaseItem**

A **Purchase** is created during checkout:

- belongs to a customer
- contains purchase items (copied from basket)
- includes total amount in EUR
- has a status lifecycle (created/paid/canceled)

This supports US_18 (purchase/check out) and US_20 (purchase history).

---

### 4.5 Payments and FX snapshot

The system supports payments for:

- rentals (US_08)
- purchases (US_19)

To ensure **financial consistency**, the model stores:

- original payment amount and currency (e.g., USD)
- the **FX conversion rate** used at the time of payment
- the computed EUR amount stored in the system

This "FX snapshot" approach guarantees:

- stable totals in history views
- auditable conversions
- independence from later exchange-rate changes

The exchange-rate provider is treated as an integration component; its output is persisted as part of payment records (not recalculated later for history).

---

### 4.6 Key invariants and constraints

These are the conceptual rules enforced by the model and services:

- **A bike's rental availability is driven by** `BikeStatus` and active rentals.
- **WaitingList is per-bike and FIFO:** entries are served in order.
- **SaleOffer is per-bike and only for eligible corporate bikes** (US_10).
- **Basket items and purchase items store price snapshots** (immutability of commercial facts).
- **Payments store FX snapshots** (immutability of financial facts).

---

## 5. Scenarios of use

**Prerequisite: Database seeding** The application includes a `DatabaseSeeder` that runs automatically on startup (in non-test profiles). It initializes the database with a consistent state for demonstration:

- **Alice (Student/Provider)**
  - Email: `alice@bike.com`
  - Password: `123456`
  - State: She has offered a bike (currently available) and has an open basket.
- **Bob (Customer)**
  - Email: `bob@bike.com`
  - Password: `123456`
  - State: He is currently on a waiting list for a corporate bike.
- **Inventory**
  - 1 corporate bike (rented by Alice).
  - 1 student bike (offered by Alice, status: `AVAILABLE`).
  - 1 corporate sale offer ("Vintage Road Bike", status: `LISTED`).

---

## S0 Authentication

To perform any operation, you must first obtain a JWT.

**Step 0.1 — Login** `POST /api/users/login`

```
curl -X POST http://localhost:8080/api/users/login \
  -H "Content-Type: application/json" \
  -d '{"email":"bob@bike.com","password":"123456"}'
```

**Result:**

- **200 OK**
- Returns a JSON object containing `accessToken`. Copy this token for subsequent requests.

---

## S1 Offer a bike for rent (US_01–US_03)

**Actor:** Alice (Student) **Goal:** Offer a personal bike for rent.

`POST /api/bikes`

**Payload:**

```
{
  "description": "Mountain Bike - Red",
  "offeredByType": "STUDENT",
  "offeredById": "<Alice-UUID-from-Token>",
  "rentalDailyRateEur": 5.00
}
```

**Result:** `201 Created`. The bike is now searchable with status `AVAILABLE`.

---

## S2 Search and rent / waiting list (US_04–US_07)

**Actor:** Bob (Customer)

**Step S2.1 — Search** `GET /api/bikes?status=AVAILABLE`

- **Result:** Bob sees Alice's "Mountain Bike".

**Step S2.2 — Rent** `POST /api/rentals`

**Payload:**

```
{
  "bikeId": 123,
  "customerId": "<Bob-UUID>",
  "days": 3
}
```

**Outcome A (bike available):**

- **201 Created**
- Response: `{"result": "RENTED", "rentalId": 55, ...}`

**Outcome B (bike unavailable):**

- **202 Accepted**
- Response: `{"result": "WAITLISTED", "waitingListEntryId": 10, "message": "Added to waiting list"}`

---

## S3 Pay rental + return with notes (US_08–US_09)

**Step S3.1 — Pay rental** `POST /api/payments/rentals`

- Allows payment in `USD`. The system converts it to `EUR` using the real-time rate and stores the FX snapshot.

**Step S3.2 — Return bike (US_09)** `POST /api/rentals/{rentalId}/return`

**Payload:**

```
{
  "authorCustomerId": "<Bob-UUID>",
  "comment": "Brakes feel a bit loose",
  "condition": "GOOD"
}
```

**Result:**

- **200 OK**

- Response includes `nextRental`. If another user was on the waiting list, this field is populated, indicating the bike was immediately assigned to them (FIFO).

---

### S4 Corporate resale flow (US_10–US_19)

**Actor:** Bob (Customer)

**Step S4.1 — Browse sales** `GET /api/sale-offers`

- Returns the "Vintage Road Bike" seeded by the corporation.

**Step S4.2 — Add to basket** `POST /api/basket/items` with `saleOfferId`.

**Step S4.3 — Check out** `POST /api/purchases/checkout`

- Converts the basket into a `Purchase` with status `CREATED`. Price snapshots are locked.

**Step S4.4 — Pay purchase** `POST /api/payments/purchases`

- Integrates with Stripe. On success, the purchase status becomes `PAID` and the sale offer becomes `SOLD`.

---

### S5 History (US_20–US_21)

**Step S5.1 — Purchase history** `GET /api/purchases`

- Lists all purchases made by the logged-in user.

**Step S5.2 — Rental history** `GET /api/rentals`

- Lists all past and active rentals for the logged-in user.

---

## 6. API overview (endpoints, security, errors)

### 6.1 Base URLs, content type, Swagger UI

**Backend base URL (dev):**

- `http://localhost:8080/api`

All controllers produce/consume JSON (`application/json`). For example, `/users` is explicitly JSON in `UserController`.

**Swagger UI (dev):**

- `http://localhost:8080/swagger-ui/index.html`

Swagger is used to expose and explore endpoints interactively (especially useful for validating request/response payloads and testing authenticated calls).

---

### 6.2 Security model (Custom JAX-RS filter)

Instead of using the heavyweight Spring Security filter chain, the application implements a lightweight, custom security layer using standard JAX-RS components:

- **Token service:** A custom `TokenService` handles the generation and validation of JSON Web Tokens (JWT). It uses **HMAC-SHA256** to sign tokens, embedding claims such as the user's ID, email, and type.
- **Authentication filter:** A JAX-RS `@Provider` (`AuthFilter`) acts as a `ContainerRequestFilter`. It intercepts all requests to endpoints annotated with `@Secured`.
    - It checks for the presence of the `Authorization: Bearer <token>` header.
    - It validates the token signature using the `TokenService`.
    - If valid, it extracts the `userId` (UUID) and injects it into the `ContainerRequestContext` property `"userId"`.
    - If invalid or missing, it aborts the request immediately with a **401 Unauthorized** status.
- **Password hashing:** User passwords are hashed using **SHA-256** (implemented in `SecurityUtils.java`). ⚠ Future improvement: migrate to bcrypt or Argon2 for enhanced security against rainbow table attacks. This is a critical security upgrade.

---

## 6.3 Endpoint catalog by controller

Below is the **functional** endpoint list, grouped by controller. (All paths are relative to `/api`.)

### 6.3.1 Users (`/users`)

- `POST /users/register` — register a user
- `POST /users/login` — login, returns JWT

### 6.3.2 Bikes (`/bikes`)

- `POST /bikes` — Offer a bike for rent (US_01–US_03). Requires a provider identity.
- `GET /bikes?status=&q=&offeredById=` — Search bikes to rent (US_04). Supports filtering by status (`AVAILABLE`), text search, or provider ID.
- `GET /bikes/all` — List all bikes regardless of status (admin/test utility endpoint).
- `GET /bikes/{bikeId}/return-notes` — View the history of return notes for a specific bike, allowing users to assess its condition before renting.

### 6.3.3 Rentals (`/rentals`)

All endpoints in this controller rely on the authenticated user's identity extracted securely from the JWT. The `userId` is obtained from the security context, preventing users from accessing or modifying rentals that do not belong to them.

- `POST /rentals` — Rent a bike or join the waiting list (US_05–US_06).
    - Returns **201 Created** (`RentResult.RENTED`) if successful.
    - Returns **202 Accepted** (`RentResult.WAITLISTED`) if the bike is unavailable.
- `POST /rentals/{rentalId}/return` — Return a bike and optionally add a condition note (US_09). This action triggers the "serve next" logic for the waiting list.
- `GET /rentals/active` — List the authenticated user's currently active rentals.
- `GET /rentals/active/bikes` — Retrieve a list of bike IDs currently rented by the authenticated user (useful for frontend checks).
- `GET /rentals/waitlist` — View the authenticated user's current status in waiting lists (US_06).

- `GET /rentals/notifications` — View notifications regarding bike availability for the authenticated user (US_07).
- `GET /rentals` — View the complete rental history for the authenticated user (US_21).

#### 6.3.4 Sales (`/sales`)

- `POST /sale-offers` — create a sale offer (US_10)
- `POST /sale-offers/notes` — add a sale note (US_11)
- `GET /sale-offers?q=` — search sale offers (US_12)
- `GET /sale-offers/{saleOfferId}` — sale offer details: price, notes, availability (US_13–US_15)
- `GET /sale-offers/by-bike/{bikeId}` — resolve offer details by bike ID

#### 6.3.5 Basket (`/basket`)

- `GET /basket` — get (or create) my open basket
- `POST /basket/items` — add sale offer to basket (US_16)
- `DELETE /basket/items/{saleOfferId}` — remove from basket (US_17)
- `DELETE /basket` — clear basket

#### 6.3.6 Purchases (`/purchases`)

- `POST /purchases/checkout` — convert open basket into a purchase (US_18)
- `GET /purchases` — purchase history (US_20)
- `GET /purchases/{purchaseId}` — purchase details

#### 6.3.7 Payments (`/payments`)

- `POST /payments/rentals` — pay rental, supports currency conversion (US_08)
- `POST /payments/purchases` — pay purchase through gateway (US_19)

(Controller descriptions explicitly tie these endpoints to gateway + conversion.)

---

### 6.4 HTTP status codes and error behavior

The controllers follow clear HTTP semantics:

- **200 OK** — successful reads, updates, and "OK" commands (e.g., basket modifications).
- **201 Created** — resource created:
  - offering bikes for rent
  - rental payment created
  - checkout creates a purchase
- **202 Accepted** — request accepted but not completed as "RENTED" (join waiting list scenario).
- **400 Bad Request** — validation / missing required data (e.g., invalid payload or missing customerId in older variants).
- **401 Unauthorized** — missing/invalid token; some controllers throw `WebApplicationException("Unauthorized", 401)` if no userId is in context.

- **404 Not Found** — referenced entity not found (bike, offer, rental, purchase), documented in Swagger annotations.
- **409 Conflict** — business rule violation (e.g., offer not available, invalid state, already paid), documented in multiple controllers.

---

### 6.5 CORS strategy (manual JAX-RS filter)

Cross-Origin Resource Sharing (CORS) is managed through a dedicated JAX-RS `ContainerResponseFilter` rather than Spring's `@CrossOrigin` annotations. This ensures consistent behavior across all JAX-RS resources.

- **Implementation:** The class `CorsFilter` is registered as a `@Provider`.
- **Allowed origin:** Explicitly set to `http://localhost:4200` to support the Angular frontend development server.
- **Preflight handling:** The filter detects HTTP `OPTIONS` requests and immediately sets the response status to `200 OK`, ensuring browsers receive the necessary permission headers before attempting actual data-modifying requests (POST, PUT, DELETE).
- **Headers injected:**
  - `Access-Control-Allow-Origin: http://localhost:4200`
  - `Access-Control-Allow-Credentials: true` (allows sending cookies/auth headers)
  - `Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS, HEAD`
  - `Access-Control-Allow-Headers: origin, content-type, accept, authorization`

---

## 7. Architecture and design

### 7.1 Architectural overview

The backend follows a **layered architecture** with clear separation of concerns:

1. **API layer (Controllers / Resources)**

- Implemented as **JAX-RS resources** (Jersey) using annotations such as `@Path`, `@GET`, `@POST`, `@Consumes`, `@Produces`.
- Responsible for:
  - HTTP routing
  - input validation (`@Valid`)
  - status code selection (e.g., 201 vs 202)
  - mapping request/response DTOs
- Example: `UserController` exposes `/users/register` and `/users/login`.

2. **Application/Use-case layer (Services)**

- Encapsulates business rules and workflows:
  - rent vs waitlist decision (US_05/US_06)
  - return flow with notes and serving the next user (US_09 + US_07)
  - corporate sale eligibility checks (US_10)

    - basket checkout (US_18)
    - payment flow orchestration (US_08/US_19)

3. **Domain layer (Entities + Enums)**

- Core business concepts (Bike, Rental, SaleOffer, Basket, Purchase, Payments, WaitingList, etc.)
- Enforces invariants through:
    - entity relationships
    - state enums (BikeStatus, RentalStatus, SaleOfferStatus, PaymentStatus)
    - persistence constraints (unique constraints, one-to-one mappings)

4. **Infrastructure layer (Persistence + external integrations)**

- PostgreSQL + JPA/Hibernate (repositories)
- Stripe gateway client
- Exchange-rate API client
- Docker runtime for local reproducibility
- Swagger/OpenAPI documentation generation

This structure keeps the API stable even if integrations evolve, and lets you test use cases at the service layer and end-to-end at the controller level.

---

## 7.2 Layer responsibilities and flow (request lifecycle)

A typical request follows this lifecycle:

1. **HTTP request enters a JAX-RS resource** Example: `POST /api/rentals` hits `RentalController`.

2. **Security is enforced (if endpoint is protected)**

- Many controllers are annotated with `@Secured`.
- A security filter/interceptor validates the JWT and injects `userId` into the `ContainerRequestContext`.

3. **Controller retrieves the current user from request context** Controllers use a helper like:

- `requestContext.getProperty("userId")` If missing, they throw 401 Unauthorized.

4. **Controller delegates to a service** Controllers remain thin: they orchestrate HTTP semantics and call service methods that implement the business behavior.

5. **Service loads entities / applies business rules**

- Repository queries load bikes/rentals/offers/baskets/purchases.
- Business rules decide:
    - rent vs waitlist (US_05/US_06)
    - sale eligibility (US_10)
    - payment admissibility (not already paid, correct state, etc.)

6. **Service calls integrations when needed**

- Stripe for payment processing (payments endpoints)
- Exchange-rate API for currency conversion (US_08/US_19)

7. **Controller returns a DTO response with HTTP status**

- `201 Created` for created resources
- `202 Accepted` for waitlist outcomes
- `200 OK` for reads/updates

---

## 7.3 Cross-cutting concerns

### 7.3.1 Security design

- **Stateless architecture:** The system relies entirely on JWTs for authentication; no server-side sessions are stored. This aligns with the REST architectural style.
- **JAX-RS-native implementation:** By implementing security via a `ContainerRequestFilter` registered in `JerseyConfiguration`, the application maintains a pure JAX-RS architecture without external security framework dependencies.
- **Context-based identity:** Controllers do not trust user input for identity (e.g., a `customerId` in a JSON body) for sensitive operations. Instead, they retrieve the authenticated user from the request context:

```
UUID userId = (UUID) requestContext.getProperty("userId");
```

### 7.3.2 Validation

Request bodies are validated using Bean Validation (`@Valid`), particularly visible in controllers such as BikeCatalog and Payments.

### 7.3.3 Consistent money handling

- The system uses **EUR as the internal consistency currency**.
- Payments accept an arbitrary currency but persist:
    - original amount/currency
    - EUR conversion result
    - conversion snapshot (FX rate used)

This supports auditability and stable history views.

### 7.3.4 State machines

The design relies on explicit states (enums) for:

- Bike availability (AVAILABLE/RENTED/...)
- Rental lifecycle (ACTIVE/CLOSED/...)
- Sale offer lifecycle (LISTED/SOLD/...)
- Payment lifecycle (PENDING/PAID/FAILED/REFUNDED)
- Purchase lifecycle (CREATED/PAID/CANCELED)

This reduces ambiguity and makes business rules enforceable and testable.

---

## 7.4 Design options and rationale (high-level)

The following design decisions shape the backend:

1. **JAX-RS (Jersey) instead of Spring MVC** Chosen to match the "Java RS" requirement and keep resources in a standard REST style.
2. **Use-case-oriented endpoints** Example: `POST /rentals` returns either RENTED (201) or WAITLISTED (202). This avoids splitting behavior across multiple endpoints and matches the user story intent.
3. **Store snapshots for commercial facts**

- Basket/Purchase items store price snapshots.
- Payments store FX snapshots. This avoids inconsistent totals between "time of browsing" and "time of payment".

4. **Request-context identity (`userId`)** Instead of requiring `customerId` in every call, "my data" endpoints derive it from authentication, improving security and simplifying frontend integration.

---

## 8. Implementation notes and integrations

> Note: configuration snippets below are described at a high level. For exact values, refer to the repository configuration files (`application.properties`, `compose.yaml`, `pom.xml`, Dockerfiles, etc.).

---

### 8.1 JAX-RS (Jersey) with Spring Boot

**Configuration:** The application is configured using `JerseyConfiguration.java`, which extends `ResourceConfig`. This class explicitly registers the components required for the REST API:

- **Resources:** Scans the `fr.eiffelbikecorp.bikeapi.controller` package.
- **Filters:** Registers `AuthFilter` for security and `CorsFilter` for frontend integration.
- **Exception mappers:** Registers custom mappers (e.g., `BusinessRuleExceptionMapper`) to translate Java exceptions into HTTP 4xx/5xx responses.
- **OpenAPI:** Registers `OpenApiResource` to generate the Swagger documentation at runtime.

---

### 8.2 Swagger / OpenAPI integration

**Goal**

Provide a **self-service API contract**:

- endpoint discovery
- schema documentation
- interactive testing (with JWT authorization)

**Implementation approach**

The project integrates Swagger/OpenAPI for JAX-RS:

- annotate controllers with:
  - `@Tag`
  - `@Operation`
  - `@ApiResponses`

- annotate DTOs with `@Schema` when needed
- declare a JWT bearer security scheme (OpenAPI `SecurityScheme`)
- attach security requirements to secured endpoints

This results in an accessible Swagger UI:

- `http://localhost:8080/swagger-ui/index.html`

This is particularly helpful because the API uses many DTOs and different HTTP status outcomes (e.g., 201 vs 202 for rent/waitlist).

---

## 8.3 Database integration (PostgreSQL)

**What is stored**

PostgreSQL stores all persistent business state:

- bike catalog + statuses
- rentals, return notes, waiting lists + notifications
- sale offers + notes
- basket, purchase, purchase items
- rental & sale payments (with Stripe reference + FX snapshot)

**Why PostgreSQL**

- strong relational integrity (constraints, foreign keys)
- reliable transaction support for state transitions (rent/return, checkout/pay)
- widely supported tooling and Docker runtime

**JPA/Hibernate mapping**

Entities map the conceptual model and typically enforce:

- one-to-one constraints (e.g., one sale offer per bike)
- uniqueness for basket/purchase item relationships
- ordering of collections (e.g., waiting list FIFO)

---

## 8.4 Docker integration

**Purpose**

Docker is used to make local development **repeatable**:

- consistent Postgres version,
- predictable credentials and ports,
- easy reset and clean environment.

**Typical workflow**

1. Start PostgreSQL:

```
docker compose up -d
```

2. Run backend locally (Spring Boot):

```
./mvnw spring-boot:run
```

Or run backend using a dev Dockerfile (if provided):

```
docker build -f Dockerfile.dev -t eiffel-bike-api:dev .
docker run --rm -p 8080:8080 --env-file .env eiffel-bike-api:dev
```

---

## 8.5 Payment gateway integration (Stripe)

**Why Stripe**

Stripe provides a robust payment gateway for:

- rental payments (US_08)
- purchase payments (US_19)

**How it is used (high-level)**

Payment endpoints are centralized under `PaymentController`.

Typical flow:

1. Validate domain state:

- rental exists and is payable
- purchase exists and is payable

2. Convert currency if needed (see next section)
3. Call Stripe to create/confirm payment
4. Persist a payment record containing:

- gateway identifier (e.g., PaymentIntent ID)
- original currency/amount
- EUR amount stored by system
- payment status (PAID/FAILED/etc.)

This design ensures:

- gateway communication is isolated to the payment use case
- the domain remains consistent even if external services fail (via explicit payment status)

---

## 8.6 Exchange-rate API integration (ExchangeRate-API)

**Configuration mapping:** The integration is configured via the `ExchangeRateApiProperties` class, which maps properties defined in `application.properties` using the prefix `fx.exchangerate`.

- `fx.exchangerate.api-key`: The secret authentication key for the external service.
- `fx.exchangerate.base-url`: The API endpoint URL (e.g., `https://v6.exchangerate-api.com/v6`).
- `fx.exchangerate.baseCode`: Set to `EUR`. This is the target currency for all conversions and the system's internal consistency currency.
- `fx.exchangerate.cache-seconds`: Defines the duration (e.g., 60 seconds) for caching fetched exchange rates. This optimization minimizes external HTTP calls, avoids rate limiting, and improves response times for the payment endpoints.

---

# 9. Tests and validation

## 9.1 Integration tests mapped to user stories

The application achieves **user story coverage** through a dedicated suite of integration tests. Each test class is annotated with `@SpringBootTest(webEnvironment = RANDOM_PORT)` and uses `TestRestTemplate` to perform end-to-end validation of the API, security, and database layers running in a Dockerized environment.

| User Story | Description | Test Class | Key Scenario Validated |
|---|---|---|---|
| **US_01** | Student offers bike | `UserStory01Test.java` | Create bike as Student → 201 Created |
| **US_02** | Employee offers bike | `UserStory02Test.java` | Create bike as Employee → 201 Created |
| **US_03** | Corp offers bike | `UserStory03Test.java` | Create bike as Corp → 201 Created |
| **US_04** | Search bikes | `UserStory04Test.java` | Filter by status/text → 200 OK with list |
| **US_05** | Rent bike | `UserStory05Test.java` | Rent available bike → 201 RENTED |
| **US_06** | Waitlist | `UserStory06Test.java` | Rent unavailable bike → 202 WAITLISTED |
| **US_07** | Notifications | `UserStory07Test.java` | Check notifications after availability change |
| **US_08** | Pay rental | `UserStory08Test.java` | Pay in USD → converted to EUR → 201 Created |
| **US_09** | Return bike | `UserStory09Test.java` | Return + note → waitlist processing triggered |
| **US_10** | Sell eligibility | `UserStory10Test.java` | Validates only rented bikes can be sold |
| **US_11** | Sale notes | `UserStory11Test.java` | Add detailed notes to sale offer |
| **US_12** | Search sales | `UserStory12Test.java` | Search active offers |
| **US_13** | View prices | `UserStory13Test.java` | Verify asking price is visible |
| **US_14** | View notes | `UserStory14Test.java` | Customer views notes on offer |
| **US_15** | Check availability | `UserStory15Test.java` | Verify SOLD items don’t appear in search |
| **US_16** | Add to basket | `UserStory16Test.java` | Add item → verifies snapshot price logic |
| **US_17** | Remove from basket | `UserStory17Test.java` | Remove item → basket is empty |
| **US_18** | Checkout | `UserStory18Test.java` | Basket → purchase (CREATED) |
| **US_19** | Pay purchase | `UserStory19Test.java` | Pay purchase in USD → converted to EUR |
| **US_20** | Purchase history | `UserStory20Test.java` | List my purchases |

| User Story | Description | Test Class | Key Scenario Validated |
|---|---|---|---|
| **US_21** | Rental history | `UserStory21Test.java` | List my rentals |

### 9.2 How to run the tests

Typical commands:

- `./mvnw test`
- or `mvn test`

Most tests are annotated with Testcontainers and are disabled automatically when Docker is not available (`@Testcontainers(disabledWithoutDocker = true)`).

---

### 9.3 Test data seeding and authentication strategy

Most tests follow the same setup strategy:

1. Seed minimal data in the database:

- create a `Customer`
- create an `EiffelBikeCorp` when needed (corporate provider/seller)

2. Generate a JWT access token via `TokenService.generateToken(customer)`
3. Call secured endpoints with `Authorization: Bearer <token>`

Example: `BikeCatalogControllerTest` creates a customer, generates a token, then calls secured endpoints using Bearer auth headers.

User registration and login are tested directly through `/api/users/register` and `/api/users/login`.

---

### 9.4 Coverage mapping (User Stories → tests → endpoints)

| User Story | What is validated | Test class | Main endpoints exercised |
|---|---|---|---|
| US_01/02/03 | Offering a bike for rent (provider creates bike) | `BikeCatalogControllerTest` | `POST /api/bikes` |
| US_04 | Search bikes available to rent (filters) | `BikeCatalogControllerTest` | `GET /api/bikes?status=&q=&offeredById=` |
| US_05/06 | Rent a bike or join waiting list | `RentalControllerTest` | `POST /api/rentals` |
| US_07 | Notifications when bike becomes available | `RentalControllerTest` | `GET /api/rentals/notifications?...` |
| US_08 | Pay rental (currency + payment method) | `RentalPaymentControllerTest` | `POST /api/payments/rentals` |
| US_09 | Return bike + attach return notes | `RentalControllerTest`, `SaleOfferControllerTest` | `POST /api/rentals/{id}/return` |

| User Story | What is validated | Test class | Main endpoints exercised |
|---|---|---|---|
| US_10/11 | Sale offer eligibility + notes | `SaleOfferControllerTest` | `POST /api/sale-offers`, `POST /api/sale-offers/notes` |
| US_12–15 | Search sale offers + view details/notes/availability | `SaleOfferControllerTest` | `GET /api/sale-offers`, `GET /api/sale-offers/{id}` |
| US_16/17 | Basket add/remove/clear | `BasketControllerTest` | `GET /api/basket`, `POST /api/basket/items`, `DELETE /api/basket/items/{id}` |
| US_18 | Checkout basket into a purchase | `PurchaseControllerTest` | `POST /api/purchases/checkout` |
| US_19 | Pay purchase through gateway (Stripe-like) | `SalePaymentControllerTest` | `POST /api/payments/purchases` |
| US_20 | Purchase history | `PurchaseControllerTest` | `GET /api/purchases` |
| US_21 | Rental history | *(implemented in controller; add test or extend existing one)* | `GET /api/rentals` (secured) |

### 9.5 Examples of key scenarios covered

**Validation errors → HTTP 400**

- Invalid bike creation request returns `400 BAD_REQUEST`.
- Invalid rent request returns `400 BAD_REQUEST`.
- Invalid rental payment request returns `400 BAD_REQUEST`.

**Business rule conflicts → HTTP 409**

- Checkout with an empty basket returns `409 CONFLICT`.
- Adding the same sale offer twice to a basket returns `409 CONFLICT`.
- Paying the same purchase twice returns `409 CONFLICT`.

**Happy-path end-to-end flows**

- Offer bike → rent → return → create sale offer eligibility flow is used inside sale/basket/purchase tests.
- Basket → checkout → pay purchase → offer becomes SOLD is checked in the purchase payment tests.

---

### 9.6 Notes / current limitations

- A few tests mention that payment method IDs may need to be realistic Stripe test values depending on how strictly the gateway is enforced (some tests use `pm_card_visa`).

---

## 10. Frontend

- **Framework:** Angular 17+ (Standalone Components)
- **State Management:** Angular Signals & Computed Properties

- **Styling:** Tailwind CSS (Utility-first framework)
- **Security:** JWT Authentication with Role-Based Access Control (RBAC)
- **Communication:** RxJS & HttpClient

### 10.1 Route Configuration

The application uses a hybrid routing strategy with a `Mainlayout` providing a persistent navigation sidebar for authorized users.

| Path | Component | Role Protection |
|---|---|---|
| `/` | `LandingComponent` | Public |
| `/login` | `Loginpage` | Public |
| `/dashboard` | `Dashboard` | Student, Employee |
| `/rentals` | `Myrentals` | Student, Employee |
| `/sales` | `MarketplaceComponent` | Student, Employee, Ordinary |
| `/offer` | `OfferBikeComponent` | Student, Employee, Corp |

### 10.2 Security: Role Guard

Access is enforced via the `roleGuard` functional guard. It intercepts navigation by:

1. **Token Check:** Verifying the existence of a JWT in `localStorage`.
2. **Base64 Decoding:** Extracting the `type` (role) from the JWT payload without a backend round-trip.
3. **Permission Check:** Matching the user role against the `allowedRoles` defined in the route data.

### 10.3 Key Component Logic

#### 10.3.1 User Service (`UserService`)

The central hub for identity. It uses **Signals** to expose the current user globally.

- It handles `login()` by storing the JWT and `setUserFromToken()` to decode user details (ID, Name, Role) for immediate UI reactivity.

#### 10.3.2 Marketplace & Basket (`MarketplaceComponent`)

Implements the resale flow (US_12–US_19).

- **Signal-based Totals:** The `totalEur` is a `computed` signal that updates automatically as the basket changes.
- **Business Rule Enforcement:** Includes logic to prevent users from adding their own listed bikes to their basket.
- **Checkout Workflow:** Transitions the user from a browsing state to a `Purchase` state, locking in the price snapshot before calling the payment API.

#### 10.3.3 Rentals & History (`Myrentals`)

Manages the university rental lifecycle (US_21).

- **Unified Dashboard:** Displays active rentals and waitlist status in one view.
- **Return Logic:** Collects bike condition and comments, which are then pushed to the backend to notify the next person on the FIFO waiting list.

### 10.3.4 Offering Flow (`OfferBikeComponent`)

The provider interface for listing inventory.

- **Conditional UI:** Shows the "List for Sale" option only for corporate bikes that meet the "previously rented" criteria.
- **Bike History:** Allows providers to see all past notes left by renters to track wear and tear.

---

## 10.4 UI/UX Implementation

### 10.4.1 Responsive Layout

The `Mainlayout` features a responsive sidebar that adapts to different screen sizes. It uses `routerLinkActive` to provide visual feedback on the current location.

### 10.4.2 Interaction Design

- **Modals & Drawers:** Used for checkout and bike details to keep the user in context.
- **Alert System:** A centralized `alert` signal provides consistent feedback for success/error states across all forms.

## 10.5 Integration Notes

### 10.5.1 Authentication Header

Most service calls utilize a private helper to inject the Bearer token:

```
private getHeaders() {
  const token = localStorage.getItem('token');
  return new HttpHeaders().set('Authorization', `Bearer ${token}`);
}
```

### 10.5.2 State Persistence

User session data is mirrored in localStorage to ensure the application remains authenticated upon browser refresh, with UserService re-initializing the signals on startup.

---

# 11. Known limitations and future work

## 11.1 Known limitations

#### 11.1.1 Error response format is not standardized

Controllers throw `WebApplicationException` for unauthorized access (e.g., when `userId` is missing in the request context). This is correct behavior, but the API would benefit from a **consistent error payload** across all endpoints (e.g., Problem Details / RFC 7807 style).

#### 11.1.2 Mixed patterns for “current customer” resolution (historical drift)

The preferred pattern is to derive customer identity from JWT (request context `userId`) used in basket/purchases/rentals “my data” endpoints. Some older patterns also appear (using `customerId` query params in some flows). This can confuse API consumers; best is to converge on **JWT identity** for all “my-*” endpoints.

#### 11.1.3 External integrations in automated tests

Payment tests use values like `pm_card_visa` and exercise purchase/rental payment endpoints. Depending on the runtime strategy, these tests may still be sensitive to gateway behavior unless Stripe/FX calls are isolated behind adapters and mocked in CI.

#### 11.1.4 Token lifecycle and session UX

JWT login is covered and returns token + expiry. However, refresh token/renewal flows are not documented as part of the API contract, which can limit longer sessions for clients.

#### 11.1.5 Concurrency edge cases are not explicitly demonstrated

Core flows like **renting**, **waitlisting**, **returning**, **checkout**, and **paying** are stateful and can face race conditions under load (two users trying to rent the same bike, two checkouts on the same offer, etc.). Integration tests validate business conflicts (409) for several cases. Still, concurrency control (locking strategy) should be explicitly documented and tested. The current implementation relies on database-level transaction isolation (default READ_COMMITTED in PostgreSQL) and optimistic locking where applicable.

### 11.2 Future work (recommended improvements)

#### 11.2.1 Standardize errors

- Implement a global exception mapper to output a uniform error schema:
    - `type`, `title`, `status`, `detail`, `instance`
- Document error codes in Swagger.

#### 11.2.2 Normalize identity handling (JWT-first)

- For endpoints returning “my data” (rent history, active rentals, basket, purchases), remove `customerId` query params and always use the authenticated `userId`.

#### 11.2.3 Improve resiliency of Stripe + FX integrations

- Wrap Stripe + FX calls behind interfaces (ports) and implement:
    - production adapters (real HTTP/Stripe)
    - test adapters (fake, deterministic)
- Add retry/circuit-breaker strategy for FX calls if needed.

#### 11.2.4 Observability

- Add structured logs for key state transitions:
    - bike status changes
    - rental creation/closure
    - waitlist enqueue/dequeue + notification creation
    - offer sold transitions

- Optional: correlation IDs per request.

#### 11.2.5 Pagination, sorting, and filtering

- Add pagination to list endpoints:
    - bikes search
    - sale offers search
    - purchase history
    - rental history

#### 11.2.6 Security enhancements

- Refresh tokens / session renewal
- Role-based access control rules made explicit (e.g., only EiffelBikeCorp can create sale offers/notes)
- Rate limiting on auth endpoints

---

## Appendix A — Runbook (commands and URLs)

This appendix is aligned with the project's current runtime configuration:

- PostgreSQL is provided via Docker Compose (`postgres:16`, DB `uge_bike`, user/pass `uge`).
- The API is configured to connect to Postgres at `jdbc:postgresql://localhost:5432/uge_bike` and uses `spring.jpa.hibernate.ddl-auto=create-drop`.
- A dev Dockerfile exists that runs the app with Maven inside a JDK 21 image.

---

### A.1 Prerequisites

**Local run (recommended):**

- Java 21
- Maven (or `./mvnw`)
- Docker + Docker Compose

**Docker run (optional):**

- Docker + Docker Compose

Project dependencies confirm you're using Spring Boot + Jersey (JAX-RS), the PostgreSQL driver, the Stripe SDK, and Swagger for JAX-RS.

---

### A.2 Start PostgreSQL (Docker Compose)

From the folder containing `compose.yaml`:

```
docker compose up -d
docker ps
```

Expected: a container named `uge_bike` exposing port `5432`.

Useful commands:

```
# Follow logs
docker logs -f uge_bike

# Stop
docker compose down

# Hard reset DB data (DANGER: deletes volume)
docker compose down -v
```

---

## A.3 Run the backend API locally (recommended)

Because your `application.properties` points to `localhost:5432`, the smoothest dev workflow is:

- Postgres in Docker
- API running locally

### A.3.1 Run

```
./mvnw spring-boot:run
# or
mvn spring-boot:run
```

### A.3.2 What to expect

- Base URL: `http://localhost:8080`
- Swagger UI: `http://localhost:8080/swagger-ui/index.html`
- DB: Postgres on `localhost:5432`, DB `uge_bike`, user/pass `uge`

### A.3.3 Important note about schema reset (test only)

`spring.jpa.hibernate.ddl-auto=create-drop` means:

- schema is created at startup
- schema is dropped at shutdown

---

## A.4 Run the backend API in Docker (optional)

You have a `Dockerfile.dev` that:

- uses `eclipse-temurin:21-jdk-alpine`
- installs Maven
- runs `mvn spring-boot:run`
- exposes port 8080

### A.4.1 Build

```
```

```
docker build -f Dockerfile.dev -t bike-api-dev .
```

#### A.4.2 Run (Mac/Windows easiest)

If Postgres is running on your host (via Compose), the container cannot reach it via `localhost`. Use `host.docker.internal` override:

```
docker run --rm -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://host.docker.internal:5432/uge_bike \
  -e SPRING_DATASOURCE_USERNAME=uge \
  -e SPRING_DATASOURCE_PASSWORD=uge \
  bike-api-dev
```

This keeps your `compose.yaml` unchanged while making the API container reach the host Postgres.

#### A.4.3 Alternative (same Docker network)

If you later decide to run **API + Postgres in the same compose project**, you can set:

- `SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/uge_bike` (where `postgres` is the service name in `compose.yaml`).

---

### A.5 External integrations configuration (FX + Stripe)

Your `application.properties` already declares:

- FX provider base URL: `https://v6.exchangerate-api.com/v6`
- FX cache: `60` seconds
- base currency code: `EUR`
- Stripe default currency: `eur`

**Security recommendation (for the report):** do not keep real API keys in `application.properties`. Prefer environment variables (or a local, uncommitted `application-dev.properties`) and load them at runtime.

Example pattern (recommended):

```
export FX_EXCHANGERATE_API_KEY="..."
export STRIPE_SECRET_KEY="..."
./mvnw spring-boot:run
```

---

### A.6 Run tests

#### A.6.1 Run all tests

```
./mvnw test
# or
```

```
mvn test
```

Your integration tests follow the Spring Boot `RANDOM_PORT` + `TestRestTemplate` pattern (end-to-end).

> If tests use Testcontainers, Docker must be running; otherwise those tests may be skipped depending on the annotations used in your suite.

---

### A.7 Quick verification URLs

- API base: `http://localhost:8080/api`
- Swagger UI: `http://localhost:8080/swagger-ui/index.html`

---

### A.8 Smoke-test with curl (minimal)

#### A.8.1 Login (get token)

```
curl -X POST http://localhost:8080/api/users/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@bike.com","password":"123456"}'
```

#### A.8.2 Call a secured endpoint (example: get my basket)

```
curl http://localhost:8080/api/basket \
  -H "Authorization: Bearer <accessToken>"
```

The basket endpoints are secured and rely on Bearer auth.

---

## Appendix B — Full endpoint reference

Swagger UI: `http://localhost:8080/swagger-ui/index.html`

---

## Appendix C — ADRs (Architecture Decision Records)

Below is a clean ADR set aligned with the backend choices (JAX-RS/Jersey + Spring Boot, Postgres, Docker, Swagger, Stripe, ExchangeRate-API, and the key domain rules).

---

### ADR-001 — Use JAX-RS (Jersey) within Spring Boot

**Context:** The project requirements emphasize a Java RS approach. At the same time, we want Spring Boot benefits (DI, configuration, testing, JPA). **Decision:** Implement controllers as **JAX-RS resources** running on **Jersey**, hosted inside **Spring Boot**. **Consequences:**

- ☑ Meets Java RS expectations (`@Path`, `@GET`, `@POST`, etc.) while keeping Spring Boot productivity.
- ☑ Controllers return `jakarta.ws.rs.core.Response`, providing explicit HTTP semantics.
- ⚠ Requires extra wiring compared to Spring MVC (resource registration, filters, OpenAPI config).

---

## ADR-002 — JWT bearer authentication; resolve identity via request context

**Context:** Many endpoints should act on "my data" (basket, purchase history, rental history). Passing customerId in query/body is error-prone and insecure. **Decision:** Use **JWT bearer tokens** for authentication. For secured endpoints, the auth layer resolves the user identity from JWT and stores `userId` into the JAX-RS `ContainerRequestContext` (e.g., `requestContext.getProperty("userId")`). **Consequences:**

- ☑ Simplifies API usage for clients (no need to send customerId everywhere).
- ☑ Reduces the risk of users accessing other users' data by guessing UUIDs.
- ⚠ Requires consistent enforcement across all secured endpoints.
- ⚠ Needs a clear strategy for token expiry and refresh (future enhancement).

---

## ADR-003 — Use-case-driven endpoint semantics for renting: 201 RENTED vs 202 WAITLISTED

**Context:** Renting a bike can either succeed immediately or enqueue the customer into the waiting list. Splitting this into two endpoints makes the client workflow more complex. **Decision:** Keep a single endpoint `POST /rentals` that returns:

- **201 Created** when the bike is rented immediately
- **202 Accepted** when the customer is placed on the waiting list

**Consequences:**

- ☑ Matches the user story mental model: "I request a rental; the system decides the outcome."
- ☑ Makes the frontend simpler (one call → interpret result).
- ⚠ Requires a clear response payload indicating `result` (e.g., RENTED/WAITLISTED).

---

## ADR-004 — Model a waiting list per bike with FIFO fairness

**Context:** The requirement demands "first come, first served" when multiple customers wait for the same bike. **Decision:** Create a **WaitingList** per **Bike**, containing ordered **WaitingListEntry** records. Entries are processed in ascending creation time (FIFO). **Consequences:**

- ☑ Clear fairness rule.
- ☑ Natural persistence model (one-to-one bike→waiting list; one-to-many waiting list→entries).
- ⚠ Concurrency must be handled carefully when returning bikes and assigning to the next customer.

---

## ADR-005 — EUR as system currency + FX snapshot persisted per payment

**Context:** Users can pay in any currency, but the system needs consistency for totals, reporting, and history. Exchange rates change over time. **Decision:** Store all domain totals in **EUR**. When a payment is made in another currency, convert to EUR and persist a snapshot containing:

- original amount + currency
- FX rate used
- computed EUR amount

**Consequences:**

- ☑ Stable purchase/rental history totals.
- ☑ Auditability (the system can explain how EUR was obtained).

- ☑ No retroactive changes in financial history.
- ⚠ Requires an external FX provider and careful rounding rules.
- ⚠ Requires cache and failure strategy if the FX service is unavailable.

---

### ADR-006 — Stripe as payment gateway

**Context:** Need a reliable gateway for card-like payments with test tooling and a mature Java SDK. **Decision:** Use **Stripe** to process rental payments and purchase payments, and store a gateway reference (e.g., PaymentIntent ID) in payment entities. **Consequences:**

- ☑ Mature SDK and sandbox testing support.
- ☑ Clear payment lifecycle and statuses.
- ⚠ Tests must isolate Stripe calls (mock adapter) or use sandbox keys carefully.
- ⚠ Keys must be externalized (env vars) to avoid leaking secrets.

---

### ADR-007 — Swagger/OpenAPI documentation for JAX-RS resources

**Context:** The API has many endpoints and DTOs; manual documentation drifts quickly. **Decision:** Generate and expose OpenAPI documentation using Swagger for JAX-RS and annotations (`@Operation`, `@ApiResponses`, `@Schema`), and publish Swagger UI at `/swagger-ui/index.html`. **Consequences:**

- ☑ Contract visibility and interactive exploration.
- ☑ Facilitates debugging and frontend integration.
- ⚠ Requires discipline to keep annotations up to date.
- ⚠ Security scheme must be documented consistently for secured endpoints.

---

### ADR-008 — PostgreSQL as the persistence store

**Context:** The domain requires relational integrity (offers, baskets, purchases, rentals, waiting lists). **Decision:** Use **PostgreSQL** with JPA/Hibernate mappings. **Consequences:**

- ☑ Strong relational model and transactional guarantees.
- ☑ Easy local reproducibility via Docker.
- ⚠ A schema migration strategy should be added (Flyway/Liquibase) for production-like evolution.

---

### ADR-009 — Docker Compose for local reproducibility

**Context:** The team needs consistent DB setup across machines and CI. **Decision:** Provide Postgres via **Docker Compose** (and optionally run the app with a dev Dockerfile). **Consequences:**

- ☑ One-command startup; stable environment.
- ☑ Easier onboarding and demos.
- ⚠ Networking differs when the app runs inside Docker vs on the host; document recommended setups.

---

### ADR-010 — Snapshots for commercial facts (basket/purchase items)

**Context:** Prices may evolve; checkout and history must remain consistent with what the buyer saw/confirmed. **Decision:** Basket items and purchase items store a **unit price snapshot in EUR** at the time of selection/checkout. **Consequences:**

- ☑ Prevents price drift issues.
- ☑ Stable purchase history totals.
- ⚠ Requires clear rules on when the snapshot is taken (add-to-basket vs checkout).

---

# Appendix D. User Manual

Welcome to the **Eiffel Bike Corp** platform. This guide provides step-by-step instructions on how to use the system based on your user profile (Student, Employee, or External Buyer).

---

## 1. Getting Started

### 1.1 Registration & Login

1. Navigate to the **Landing Page** (`/`).
2. Click **Join Now** to create an account.
3. Select your profile type:
    - **Student/Employee:** Required for renting bikes and offering bikes.
    - **Ordinary:** For external users who only wish to purchase corporate bikes.
4. Once registered, log in via the **Login** page to receive your access token.

---

## 2. For University Members (Students & Employees)

### 2.1 Renting a Bike

1. Go to the **Dashboard** (Rent a Bike).
2. Browse the list of available bikes. You can filter by description or daily rate.
3. Click **Rent Now**.
    - **If Available:** Your rental starts immediately.
    - **If Unavailable:** You will see an option to **Join Waiting List**. You will be notified when the bike is returned.

### 2.2 Managing Your Rentals

1. Navigate to **My Rentals**.
2. **Active Rentals:** View the bikes you currently have.
3. **Returning a Bike:** * Click the **Return** button.
    - Select the bike's condition (Good, Fair, Poor).
    - Add a mandatory note describing any issues (e.g., "Chain is squeaky").
4. **Waitlist:** Check your position in the queue for bikes you are waiting for.

### 2.3 Offering Your Bike for Rent

1. Navigate to **Offer a Bike**.
2. Fill out the form:
    - **Description:** Brand, color, and model.
    - **Daily Rate:** Set your price in EUR.
3. Click **Submit**. Your bike is now visible in the catalog for other members to rent.

---

## 3. For Buyers (All Users)

### 3.1 Browsing the Marketplace

1. Go to **Buy a Bike** (Marketplace).
2. Search for used corporate bikes listed by Eiffel Bike Corp.
3. Click **View Details** to see the bike's rental history and condition notes.

### 3.2 Shopping Basket & Checkout

1. Click **Add to Basket** on a bike you want.
2. Open your **Basket** (right-side drawer).
3. Review the total price. Note: The price is locked in EUR.
4. Click **Checkout**.
5. Enter your payment details:
    - Select your preferred **Currency** (USD, GBP, etc.).
    - The system will show the real-time conversion rate.
6. Click **Confirm Payment**. Once processed, the bike is yours!

---

## 4. For Corporate Administrators

### 4.1 Selling Corporate Bikes

1. Go to **Offer a Bike**.
2. View the list of company-owned bikes.
3. If a bike has been rented at least once, the **List for Sale** button will appear.
4. Set an **Asking Price** and add detailed sales notes before listing it on the public marketplace.

---

## 5. Frequently Asked Questions (FAQ)

> **Q: Can I buy my own bike?** > A: No. The system prevents users from adding their own listed bikes to their shopping basket.
>
> **Q: How does the waiting list work?** > A: It follows a First-In, First-Out (FIFO) rule. The first person to join the list gets the bike the moment the previous renter returns it.
>
> **Q: Can I pay in USD?** > A: Yes. While the system operates in EUR, the payment gateway handles conversion automatically using a real-time FX snapshot.

---

## 6. Support & Troubleshooting

- **Token Expired:** If you are suddenly redirected to the login page, your session has expired. Please log in again.
- **Unauthorized Access:** If a menu item is missing, your user profile (e.g., Ordinary) may not have permission to access that feature.

---

### References

https://medium.com/@nolomokgosi/basics-of-architecture-decision-records-adr-e09e00c636c6

https://www.atlassian.com/agile/project-management/user-stories

https://medium.com/@theopendle/creating-a-jersey-api-with-spring-boot-87a1af0512e5