

Relatório do Projeto UrbanWheels - Grupo 13

Guilherme Soares (fc62372), Vitória Correia (fc62211), Duarte Soares (fc62371)

Semestre 2025/2026

1 Introdução

O presente relatório descreve o desenvolvimento do backend do sistema **UrbanWheels**, uma plataforma de bicicletas partilhadas para a empresa fictícia ConcreteCast Solutions. O objetivo do sistema é gerir utilizadores, bicicletas, estações e viagens de forma eficiente e segura.

Observação sobre commits: Durante o desenvolvimento, alguns commits foram registados usando os nomes de utilizador habituais do GitHub em vez dos utilizadores institucionais (guimbreon para Guilherme Soares e vitoriateixeiracorreia para Vitória Correia).

2 Arquitetura do Sistema

O sistema foi desenvolvido em **Spring Boot** utilizando Java e a arquitetura está organizada em camadas:

- **Controller:** expõe os endpoints REST e recebe pedidos HTTP.
- **Service:** contém a lógica de negócio.
- **Repository:** gerencia acesso à base de dados usando JPA/Spring Data.

2.1 Design Orientado a Domínio

O modelo de domínio foi mapeado a partir das tabelas SQL fornecidas. As principais entidades são:

- **User** (abstract), com subclasses **Client** e **Admin**.
- **Station**, representando as estações de bicicletas.
- **Bike** e **BikeState**, representando a frota e seus estados.
- **Trip**, representando viagens realizadas pelos clientes.
- **Maintenance**, representando operações de manutenção realizadas por administradores.

2.2 Extensibilidade para WeatherWise

O sistema foi projetado para permitir uma futura integração com o WeatherWise, de modo que:

- A entidade **Station** possui um campo opcional `weatherConditionId`, que permitirá associar uma previsão meteorológica futura à estação. Este campo será preenchido com o `id` da condição meteorológica correspondente da entidade **WeatherCondition**.
- O endpoint REST `GET /stations` pode ser facilmente atualizado para retornar informações meteorológicas, como a condição do tempo associada a cada estação, ao incluir o `weatherConditionId` na resposta.
- O **WeatherController** fornece dois endpoints para interação com o serviço meteorológico:
 - `GET /weather/current`: Este endpoint retorna a condição climática atual para as coordenadas geográficas fornecidas (latitude e longitude).
 - `GET /weather/history`: Este endpoint retorna o histórico de condições meteorológicas para as coordenadas fornecidas.
- A lógica de integração com o WeatherWise será desacoplada da lógica de negócio principal. A **WeatherService** é responsável por obter as condições meteorológicas atuais e históricas. Neste momento, os métodos da **WeatherService** retornam dados mock, mas a integração com um provedor de dados real será adicionada no futuro.
- A classe **WeatherCondition** representa as condições meteorológicas e pode ser expandida para incluir mais detalhes, como temperatura, umidade, ou outros parâmetros meteorológicos relevantes. A interface **WeatherService** será implementada para consultar um serviço externo e preencher essas informações.
- A comunicação com o WeatherWise será feita por meio de um serviço dedicado, garantindo que a integração não afete o desempenho e a segurança do sistema.

Este design permite que a funcionalidade de previsão do tempo seja integrada de forma escalável e sem impactos negativos nas funcionalidades existentes.

3 Mapeamento JPA e Justificações

3.1 Estratégia de Herança (Entidades User, Client e Admin)

A decisão de mapeamento mais significativa reside na hierarquia de utilizadores:

- **Entidade Base: User**
- **Mapeamento:** A classe `User` é definida como uma classe abstrata (`public abstract class User`) e é mapeada para a tabela principal `users`.
- **Decisão de Herança:** `@Inheritance(strategy = InheritanceType.JOINED)`.
- **Justificação:** Esta estratégia cria uma tabela separada para cada classe (`users`, `client`, `admin`). Esta é a abordagem mais normalizada (3^a Forma Normal).

Vantagens:

- **Integridade e Unicidade:** Permite que as colunas comuns e obrigatórias, como `email` (único e não-nulo) e `name` (não-nulo), sejam armazenadas apenas na tabela `users`, evitando duplicação e garantindo que cada utilizador, independentemente do seu tipo, tem um email único.
- **Flexibilidade:** As tabelas `client` e `admin` armazenam apenas os atributos específicos, mantendo o modelo de dados limpo.

Entidades Filhas: Client e Admin

- **Mapeamento:** As classes `Client` e `Admin` estendem `User` e são mapeadas para as suas próprias tabelas (`client` e `admin`).
- **Justificação:** Representam especializações do conceito `User` com relações distintas: um `Client` está associado a uma `Subscription` e a `Trips`, enquanto um `Admin` está associado a `Maintenances`.

3.2 Mapeamento de Chaves Primárias e Colunas

- **Chaves Primárias (IDs):** Todas as entidades utilizam `@Id` e `@GeneratedValue(strategy = GenerationType.IDENTITY)`.
- **Justificação:** `GenerationType.IDENTITY` é o método preferido em bases de dados modernas (como PostgreSQL, utilizado no projeto) para usar a funcionalidade de auto-incremento da própria base, otimizando a inserção de novos registo.

Restrições de Coluna (@Column):

- `nullable = false`: Aplicado a campos essenciais como `User.email`, `Station.name`, `Station.lat`, `Station.lon`, `Station.maxDocks`, `Bike.model` e `Trip.startTime`. Isto impõe regras de negócio vitais: estas informações devem estar presentes para um registo ser válido.
- `unique = true`: Aplicado a `User.email`, `Subscription.name` e `State.description`, garantindo a unicidade lógica e a prevenção de dados duplicados.

3.3 Mapeamento de Relações

As relações refletem o fluxo da aplicação:

- **Bicicleta e Estado:** `Bike.state` (`ManyToOne`, `nullable = false`) - Cada bicicleta deve estar num estado definido (ex: "Disponível", "Em Uso", "Em Manutenção"). A restrição `nullable = false` garante essa regra.
- **Bicicleta e Estação:** `Bike.station` (`ManyToOne`, opcional) e `Station.bikes` (`OneToMany`) - Uma bicicleta pode ou não estar numa estação (se estiver a ser utilizada, estará fora de uma estação). A relação `ManyToOne` em `Bike` não tem a restrição `nullable = false`, permitindo que o campo da chave estrangeira seja nulo.

- **Cliente e Subscrição:** Client.subscription (ManyToOne, nullable = false, FetchType.EAGER) - Um cliente deve ter um tipo de subscrição (nullable = false). O uso de FetchType.EAGER indica que a informação da subscrição é sempre carregada em conjunto com o objeto Client.
- **Viagens:** Trip.bike, Trip.user, Trip.startStation (ManyToOne, optional = false/nullable = false) - Uma viagem deve ter uma bicicleta, um utilizador e uma estação de partida.
- **Viagens (Fim):** Trip.endStation (ManyToOne, opcional) - A estação de destino é opcional, pois só é preenchida quando a viagem é concluída.
- **Manutenção:** Maintenance.bike, Maintenance.admin (ManyToOne, optional = false) - Um registo de manutenção deve estar sempre ligado a uma bicicleta e a um administrador.

3.4 Mapeamento Bidirecional

A maioria das relações é mapeada de forma bidirecional (*ex: Bike tem uma lista de Trips e Trip tem uma referência para a Bike*). O lado proprietário da relação (@ManyToOne) contém a chave estrangeira na base de dados (*ex: Trip.bike gera a coluna bike_id na tabela trip*). O lado inverso (@OneToOne) utiliza o atributo mappedBy (*ex: Bike.trips usa mappedBy = "bike"*), indicando que a chave de mapeamento é definida na outra classe. Esta é uma prática padrão de JPA/Hibernate para manter a consistência do modelo.

3.5 Decisões de Design para Consistência de Dados

Para alguns casos de uso, foi necessário garantir que certos valores já existam na base de dados(devido ao referido acima, sobre o /data) antes de serem atribuídos:

- **Alterar estado de uma bicicleta (Use Case F):** Ao reportar que uma bicicleta precisa de manutenção ou que já saiu da manutenção, o novo estado (BikeState) deve já existir na base de dados. Para suportar isto, foram adicionados endpoints REST para listar todos os estados existentes (GET /bike-states) e para inserir novos estados (POST /bike-states), garantindo consistência e evitando inserções inválidas.
- **Registrar um novo utilizador (Use Case H):** Cada cliente deve ter uma subscrição (subscription) válida, que também deve existir previamente na base de dados. Para este fim, foram criados endpoints REST para listar todas as subscrições disponíveis (GET /subscriptions) e para adicionar novas subscrições (POST /subscriptions). Isso assegura que apenas subscrições válidas sejam atribuídas a novos clientes.

Dessa forma, o sistema mantém integridade referencial e evita erros ao criar ou atualizar entidades dependentes de valores pré-existentes na base de dados.

4 Diagramas do Sistema

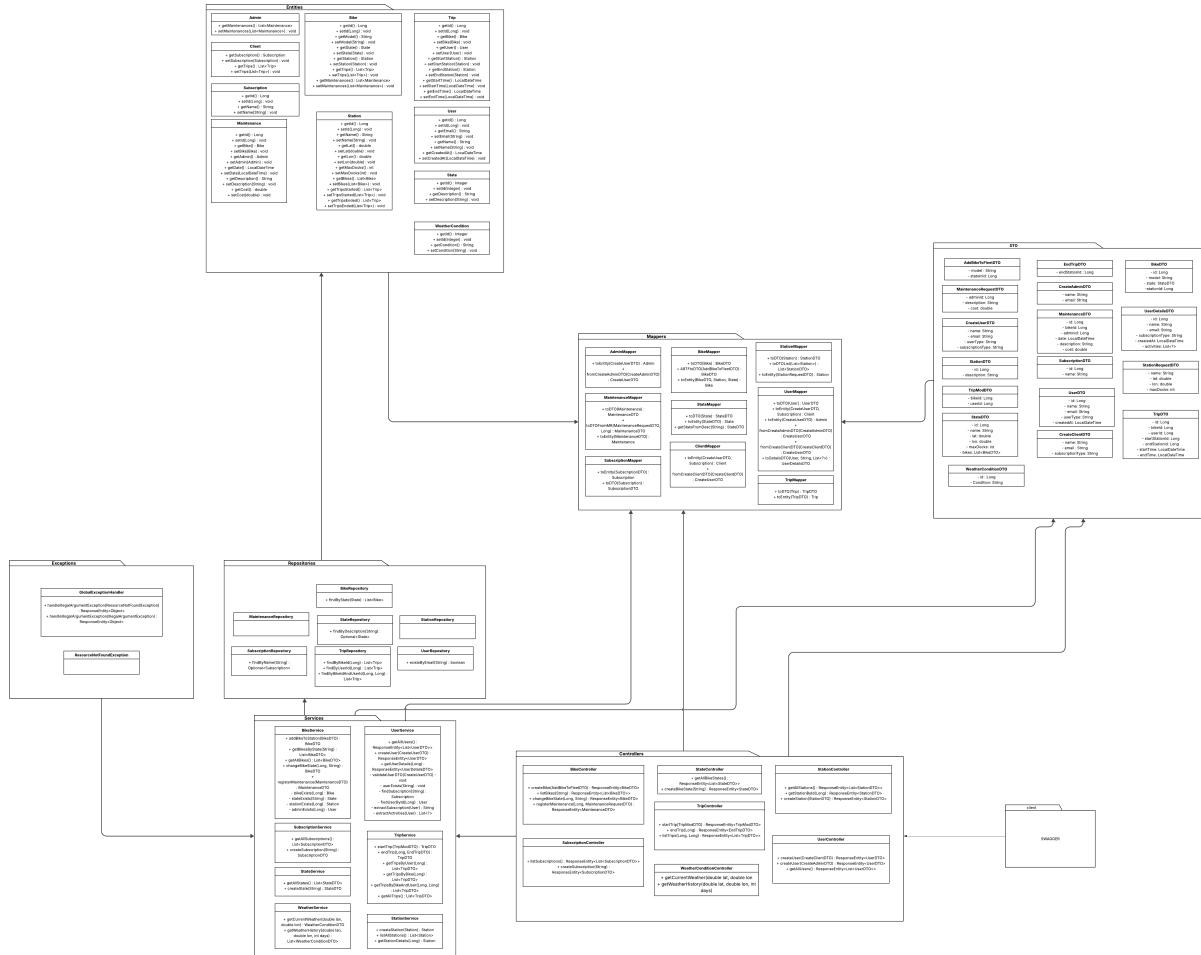


Figure 1: Diagrama de classes do UrbanWheels.

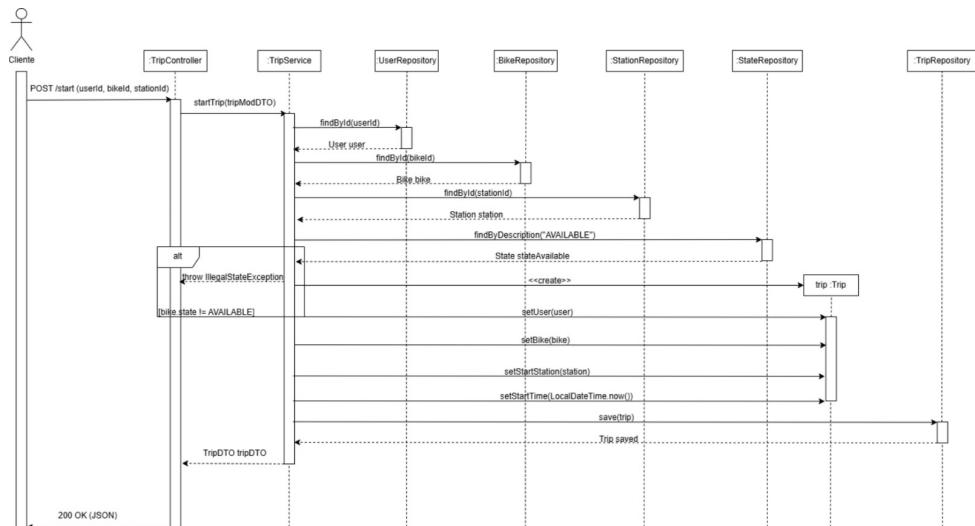


Figure 2: Diagrama de sequência para o caso de uso Levantar Bicicleta.

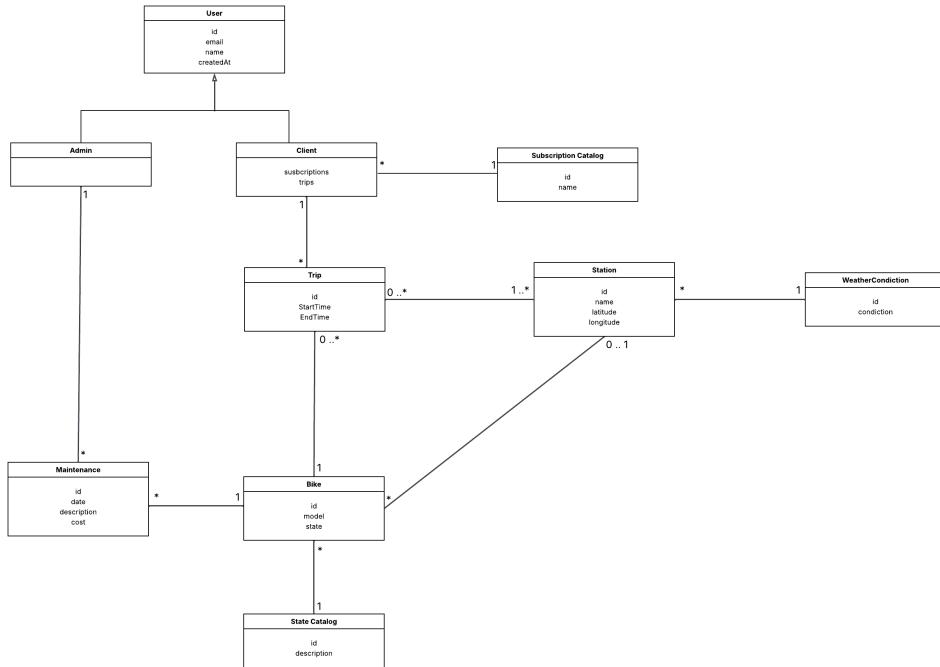


Figure 3: Modelo de Domínio.

Os diagramas poderão ser vistos no final do documento em maior escala.

5 Gestão de Utilizadores e Funções (User, Client, Admin)

A arquitetura de utilizadores suporta um sistema de duas camadas (Clientes e Administradores) que partilham dados de identidade, mas têm responsabilidades distintas:

5.1 Identidade e Unicidade

O campo `email` na classe base **User** é definido como único e não-nulo (`unique = true`, `nullable = false`).

Decisão de Negócio: Garante que cada pessoa no sistema (seja cliente ou administrador) tem uma conta distinta, essencial para a autenticação e prevenção de contas duplicadas.

5.2 Subscrição Obrigatória para Clientes

A entidade **Client** tem uma relação `ManyToOne` com **Subscription** definida como não-nula (`nullable = false`).

Decisão de Negócio: Impõe que nenhum cliente pode existir sem estar ligado a um tipo de subscrição. Isto é crucial para validar o acesso ao serviço de aluguer de bicicletas e para o modelo de faturação.

5.3 Separação de Funções

A utilização da herança JOINED para Client e Admin permite que cada função tenha os seus atributos e relações específicas (ex: Client para Trips e Admin para Maintenances), garantindo que as permissões e dados relevantes são distintos.

6 Gestão da Frota (Bike, State, Maintenance)

A gestão da frota é concebida para fornecer total visibilidade sobre o inventário, condição e histórico de serviço de cada bicicleta.

6.1 Estado Obrigatório da Bicicleta

O campo Bike.state é uma relação ManyToOne não-nula.

Decisão de Negócio: Uma bicicleta tem de estar sempre num estado conhecido (ex: "AVAILABLE", "IN_USE", "MAINTENANCE"). Isto é essencial para o sistema, para decidir se uma bicicleta pode ser alugada ou se precisa de intervenção.

6.2 Rastreabilidade da Manutenção

A entidade Maintenance é obrigatória e deve incluir a bicicleta, o administrador responsável, a descrição e o custo.

Decisão de Negócio: Garante a responsabilização e o registo financeiro completo do ciclo de vida de cada bicicleta. Permite calcular os custos operacionais e rastrear a qualidade do serviço.

7 Logística e Localização (Station)

O mapeamento das estações reflete a capacidade finita e a localização geográfica como fatores chave.

7.1 Localização Essencial

Os atributos lat e lon (latitude e longitude) são não-nulos.

Decisão de Negócio: As estações precisam de ter coordenadas geográficas para no futuro ser feita uma integração entre o urbanwheels e o weatherwise.

7.2 Capacidade Fixa

O campo maxDocks é não-nulo.

Decisão de Negócio: Define a restrição de capacidade operacional da estação. Este valor é usado para determinar se um cliente pode devolver uma bicicleta.

7.3 Localização Dinâmica da Bicicleta

A relação Bike.station é nula por omissão (não tem nullable = false).

Decisão de Negócio: Uma bicicleta só está numa estação quando está atracada e disponível. Se for nula, a bicicleta está em trânsito/em utilização por um cliente.

8 Ciclo de Vida da Viagem (Trip)

O design da entidade Trip é a decisão central que modela o estado de um aluguer:

8.1 Início de Viagem Obrigatório

Os campos `bike`, `user`, `startStation` e `startTime` são não-nulos.

Decisão de Negócio: Uma viagem só pode começar se soubermos quem a fez, com que bicicleta, onde começou e quando.

8.2 Fim de Viagem Opcional (Modela o Estado)

Os campos `endStation` e `endTime` são nulos por omissão (não têm restrições de não-nulo).

Decisão de Negócio:

- Se ambos forem nulos, a viagem está em andamento/ativa (a bicicleta está fora da estação).
- Se ambos forem preenchidos, a viagem está concluída e pronta para faturação.

Esta decisão é a base da regra de negócio para determinar as viagens pendentes.

9 Conclusão

O backend do UrbanWheels foi desenvolvido com foco em regras de negócio, integridade de dados e extensibilidade futura. O uso de JPA, DTOs e arquitetura em camadas garante manutenção simples e facilita futuras integrações, como a do WeatherWise, sem comprometer o funcionamento atual do sistema.

