Desenvolvimento de uma arquitetura de backend para um serviço de transporte de pessoas através de bicicletas.

Trabalho de Conclusão de Curso

Rodrigo Silva de Oliveira

Manoel Carvalho Marques Neto
Orientador

Instituto Federal da Bahia - IFBA
Curso de Análise e Desenvolvimento de Sistemas
Campus Salvador

Salvador, Bahia, Brasil
05 de maio de 2025

Sumário

1 Visão Geral

1.1 Declaração do Problema

Atualmente, os principais meios de transporte de pessoas oferecidos por aplicativos, são por condução através de carros ou motos. Com o meu trabalho, estou trazendo uma nova proposta, ousada e inovadora para o trasporte de pessoas solicitadas por aplicações de software. Essa nova proposta será oferecida através da locomoção por bicicletas, onde um cliente solicitará uma corrida e um ciclista cadastrado na plataforma realizará o deslocamento do cliente até o ponto desejado. A nova prosta de deslocamento de pessoas através de bicicletas, foca em distancias curtas e medias, trazendo benefícios tanta para o cliente como para o ciclista. Com o deslocamento através de bicicletas, o cliente poderá ter um custo menor no valor da corrida e ainda evitar passar por congestionamento no trânsito, contribuindo para o fluxo de pessoas nas ruas como um todo. A proposta também oferece novas oportunidades aos ciclistas, pois eles poderão se cadastrar na plaforma e realizar o serviço de trasporte de pessoas.

1.2 Proposta de Solução de Software

Para solucionar o problema apresentado, será desenvolvido uma arquitetura de backend, moderna, seguindo boas práticas de desenvolvimento e divisão de responsabilidades. A arquitetura seguirá o modelo de microsserviços, onde podemos ter aplicações independentes, com configurações proprias, executando suas tarefas de forma autonoma. Cada microsserviço ficará responsável por um dominio da arquitetura, possuindo código e base de dados propria. Esse tipo de arquitetura nos permite diversas vantagens como:

a) Escalabilidade: microsserviços individuais podem ser escalados de forma independente com base na sua demanda.

b) Disponibilidade da arquitetura: Caso um serviço apresente falha, não afetará diretamente o outro serviço.

c) Flexibilidade de tecnologia: Cada serviço poderá ser construído com uma tecnologia diferente e possuir dependencias especificas.

d) Velocidade na entrega: serviços menores podem facilitar o desenvolvimento e teste de novas features, possibilitando a entrega em produção de forma agil.

1.3 Tecnologias da Solução

Para o desenvolvimento do sistema de transporte de pessoas através de bicicletas, foram escolhidas as seguintes tecnologias:

1) Spring Boot, foi o framework escolhido para o desenvolvimento dos microsserviços, por ser um framework moderno e bastante utilizado no mercado. O spring oferece facilidade e velocidade no desenvolvimento de aplicações, permitindo iniciar a construção de um projeto de forma simples e com pouca configuração manual. Ele

oferece suporte a diversas bibliotecas utéis, como banco de dados relacionais e não relacionais, mecanismo de cache, documentação de APIs, segurança, autenticação e autorização, comunicação assincrona com sistemas de mensagerias como o Apache Kafka e varias outras funcionalidades essenciais para o desenvolvimento de uma aplicação(Spring.io, 2025).

2) Kotlin foi escolhida por ser uma linguagem de programação simples, moderna e orientada a objetos, facilitando a escrita, a legibilidade do código e a criação de testes. Um de seus principais diferenciais é a segurança em relação a referências nulas: se o compilador identificar que uma variável não nula está recebendo um valor nulo, ele impedirá essa atribuição, garantindo maior segurança para o código em tempo de execução(Kotlinlang, 2025).

3) Gradle é uma ferramenta de build, utilizada para compilar, testar, empacotar e gerenciar as dependências do projeto.

4) Redis, utilizado como cache para as aplicações, é um banco de dados em memória que armazena valores no formato chave-valor. Ele melhora significativamente a performance das aplicações, pois sua latência de leitura e escrita é inferior a milissegundos.

5) Banco de dados, o banco de dados escolhido foi o PostgreSQL, por ser de código aberto e amplamente reconhecido no mercado. O PostgreSQL armazena dados em tabelas, organizando os registros em linhas e colunas, e é totalmente compatível com a linguagem SQL. Trata-se de um banco de dados compatível com diversos sistemas operacionais e com uma ampla gama de linguagens de programação, como Java e Kotlin. Por estar consolidado no mercado há muitos anos, oferece segurança, confiabilidade e integridade dos dados(Postgresql.org, 2025).

6) Kafka, sistema de mensageria open source utilizado para comunicação assíncrona e em tempo real. O kafka é um sistema distribuido e tolerante a falhas, onde podemos publicar mensagens em topicos e adicionar ouvintes para escutar as mensagens e realizar o processamento necessário.

7) IntelliJ IDEA, uma IDE de desenvolvimento que possui uma versão gratuita, foi escolhido por ser bastante amigável ao desenvolvimento de microsserviços em Kotlin. A IDE oferece diversos recursos que facilitam o processo de desenvolvimento, como autocomplete, ferramentas de pesquisa, build e execução de testes.

8) Logs, utilizados para fornecer uma rastreabilidade das operações que estão acontecendo nos microsserviços. Serão gravados logs nos níveis de informação, warning e erros. Os logs são muito importantes pois demostram em tempo real o estado da aplicação, e também pode ser utilizado para consultas futuras.

9) Kotest, framework de teste, utilizado para criar e executar os testes de unidade na aplicação. Ele oferece diversos estilos de escrita facilitando seu uso por parte do desenvolvedor.

2 Requisitos

2.1 Requisitos Funcionais

RF1 Como um usuário, eu quero realizar login no sistema para acessar as funcionalidades disponíveis no software.

RF2 Como um usuário, eu quero cadastrar meus dados pessoais no software para ter uma melhor experiência.

RF3 Como um usuário, eu quero atualizar minhas informações pessoais no software para manter sempre meus dados consistentes.

RF4 Como um usuário, eu quero listar todas as corridas realizadas por mim para ter uma visibilidade de como está meu desempenho.

RF5 Como um usuário, eu quero avaliar a corrida realizada para garantir maior segurança e confiança no uso da aplicação.

RF6 Como um cliente, eu quero solicitar o cancelamento de uma corrida para ter uma melhor experiência caso eu desista de realizar meu deslocamento.

RF7 Como um ciclista, eu quero solicitar o cancelamento de uma corrida para ter uma melhor experiência caso eu desista de realizar o deslocamento do cliente.

RF8 Como um ciclista, eu quero aceitar, iniciar e finalizar uma corrida para contribuir com uma melhor experiência para o cliente.

RF9 Como um cliente, eu quero solicitar uma corrida para realizar meu deslocamento de um ponto a outro.

2.2 Requisitos Não-Funcionais

RNF1 O sistema deve realizar autenticação e autorização de usuários.

RNF2 O sistema deve possuir uma latência de no máximo 1 segundo.

RNF3 O sistema deve realizar conexão com banco de dados relacional postgres.

RNF4 O sistema deve possuir comunicação com kafka para executar operações de mensageria.

RNF5 O sistema deve registrar logs de erros e informações importantes para auditoria.

RNF6 O sistema deve ser construído seguindo a arquitetura de microsserviços e divisão de responsabilidades.

RNF7 O sistema deve ser estruturado seguindo o padrão de Arquitetura Hexagonal.

RNF8 O sistema deve possuir um mecanismo de cache de dados para reduzir a latência das operações.

RNF9 O sistema deve fornecer a documentação dos endpoints disponíveis através do Swagger.

RNF10 O sistema deve permitir ser implantado através de containers docker.

RNF11 O sistema deve ser construído com o framework Spring boot, liguagem Kotlin e gradle para gereniamento de dependências.

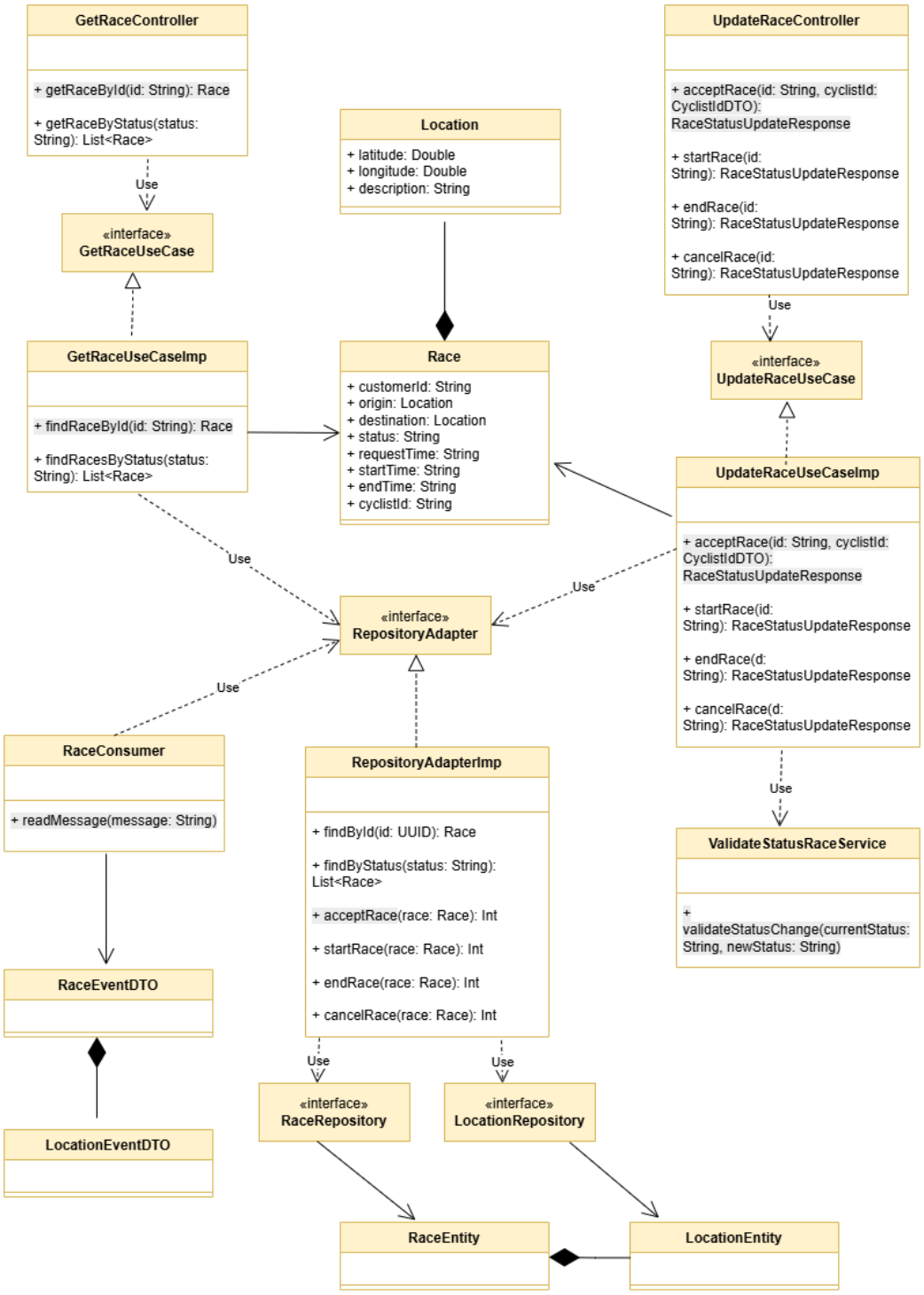
RNF12 O sistema deve possuir testes de unidade com cobertura de código de no mínimo 75%.

RNF13 O sistema deve possuir um template específico de retorno para os casos de erro e também para o casos em que será necessário informar algo sobre a requisição.

3 Design

3.1 Projeto UML

3.1.1 Diagrama de classes

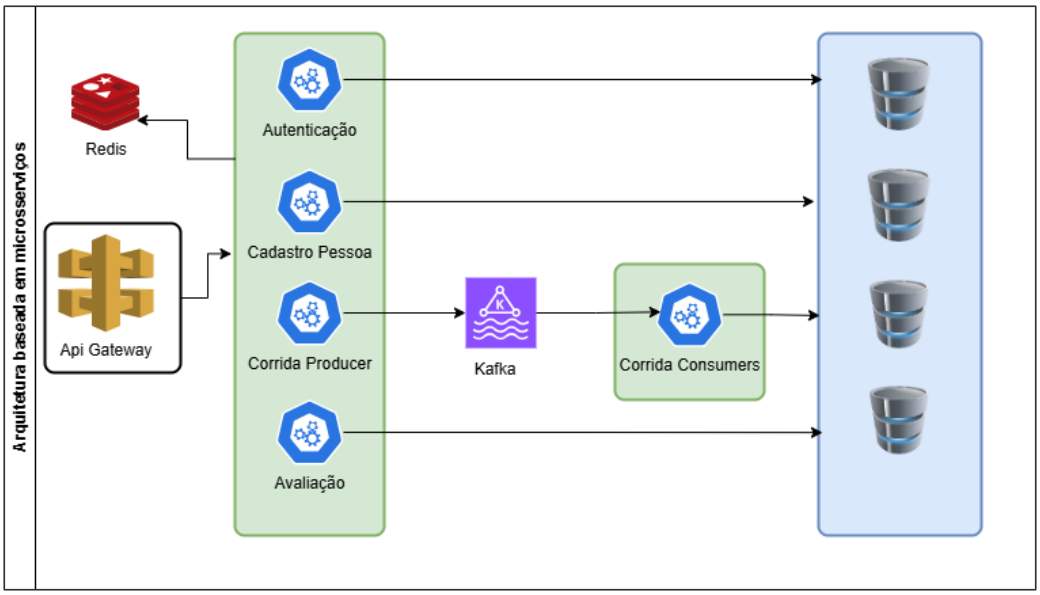


Representação do fluxo de leitura de dados do kafka e operações de CRUD sobre a corrida.

3.1.2 Diagrama de casos de uso

Casos de usos para cliente e ciclista.

3.2 Visão Arquitetural



3.3 Modelo de Banco de Dados

3.3.1 Modelo lógico de dados

O objetivo do modelo de dados lógico é representar as entidades, atributos e suas relações. Ele exibe uma visão abstrata de como os dados se organizam e se relacionam no sistema. Sua implementação é independente de banco de dados, fornecendo uma visão futura para a criação do modelo físico.

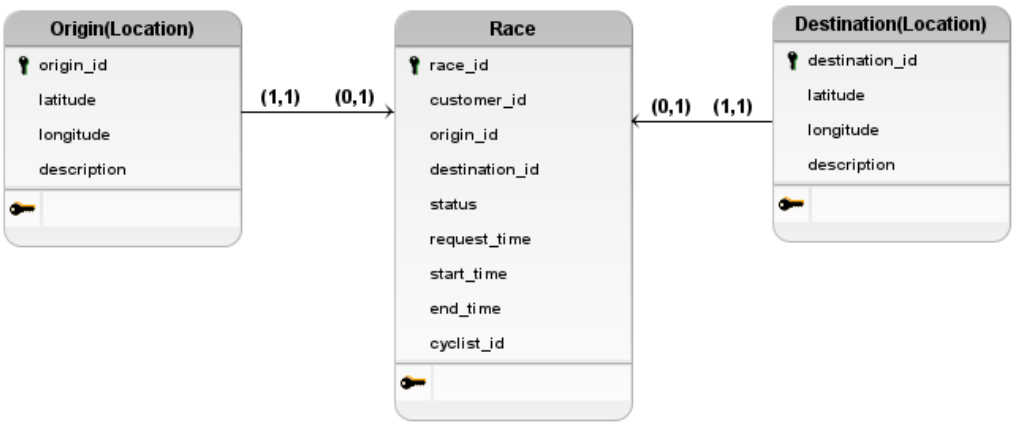


Diagrama lógico referente ao banco de dados de corrida.

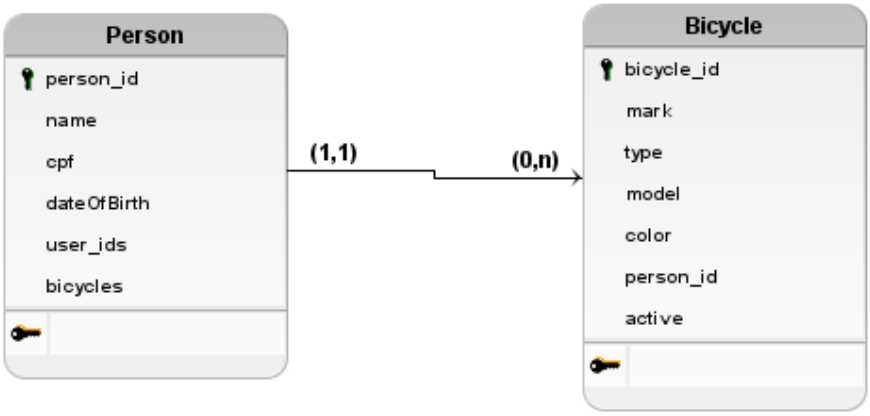


Diagrama lógico referente ao banco de dados de gerenciamento de pessoas.

Diagrama lógico referente ao banco de dados de autenticação e gerenciamento de usuários.

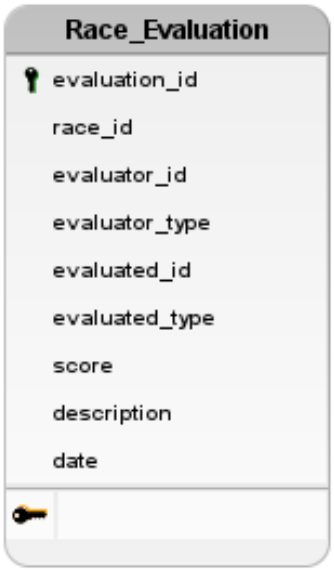


Diagrama lógico referente ao banco de dados de avaliação e comentários da corrida.

3.3.2 Modelo físico de dados

O modelo físico de dados define a forma concreta como os dados são armazenados no sistema. Ele especifica os tipos de dados, as restrições e é projetado para um sistema de gerenciamento de banco de dados específico.

Para o sistema de transporte de pessoas através de bicicletas, foram escolhidas as seguintes tabelas:

1) **TB_RACE**: Tabela responsável por armazenar e gerenciar as informações da corrida. Seus atributos são:
    1) **race_id**: Armazena o id da corrida, no formato UUID. É um campo obrigatório e não aceita valor nulo.
    2) **customer_id**: Armazena o id do cliente que está solicitando a corrida. É um campo obrigatório e não aceita valor nulo.
    3) **cyclist_id**: Armazena o id do ciclista que aceita a corrida e realiza o deslocamento do cliente entre dois pontos. É um campo do tipo UUID e permite valor nulo.
    4) **end_time**: Armazena a data e hora de finalização da corrida. Seu tipo é character e permite o valor nulo.
    5) **request_time**: Armazena a data e hora que a corrida foi solicitada. Seu tipo é character. É um campo obrigatório, e não aceita valor nulo.
    6) **start_time**: Armazena a data e hora que a corrida foi iniciada. Seu tipo é character. Permite o valor nulo.
    7) **status**: Armazena o status atual da corrida. Os valores permitidos são, SOLICITADA, A_CAMINHO, INICIADA , FINALIZADA, CANCELADA. Seu tipo é character. É um campo obrigatório e não aceita valor nulo.

8) **destination_id**: É um campo de chave estrangeira para a tabela de localização. Armazena o id do local de destino desejado pelo cliente. É um campo obrigatório, do tipo UUID e não aceita valor nulo.

9) **origin_id**: É um campo de chave estrangeira para a tabela de localização. Armazena o id da localização inicial do cliente. É um campo obrigatório, do tipo UUID e não aceita valor nulo.

2) **TB_LOCATION**: Tabela resposável por armazenar as coordenadas geográficas e uma descrição do local de inicio e fim da corrida solicitada pelo cliente. Seus campos são:

1) **location_id**: Armazena o id da localização, no formato UUID. É um campo obrigatório e não aceita valor nulo.

2) **description**: Armazena a descrição da localização no formato de string. É um campo obrigatório e não aceita valor nulo.

3) **latitude**: Armazena a coordenada geografica de latitude em relação a localização. É um campo obrigatório do tipo double e não aceita valor nulo.

4) **longitude**: Armazena a coordenada geografica de longitude em relação a localização. É um campo obrigatório do tipo double e não aceita valor nulo.

3) **TB_PERSON**: Tabela resposável por armazenar e gerenciar as informações da pessoa. Armazena dados de clientes e ciclistas. Seus campos são:

1) **person_id**: Armazena o id unico da pessoa. É um campo obrigatório do tipo UUID e não aceita valor nulo.

2) **cpf**: Armazena o cadastro de pessoa fisica(cpf). É um campo obrigatório, unique e não aceita valor nulo.

3) **date_of_birth**: Armazena a data de nascimento da pessoa. É um campo obrigatório do tipo character e não aceita valor nulo.

4) **name**: Armazena o nome da pessoa. É um campo obrigatório do tipo character e não aceita valor nulo.

4) **TB_PERSON_USER_ID**: Tabela resposável por armazenar a relação de pessoas e seus usuários. O sistema permite que uma pessoa possa ter um perfil de clista e outro de cliente. Seus campos são:

1) **person_id**: É um campo de chave estrangeira para a tabela de pessoa referenciando o id da pessoa. É obrigatório e não aceita valor nulo.

2) **user_id**: Armazena o id do usuário logado. É um campo obrigatório, do tipo UUID e não aceita valor nulo.

5) **TB_BICYCLE**: Tabela resposável por armazenar e gerenciar as informações das bicicletas. Um ciclista pode ter uma ou mais bicicletas. Seus campos são:

1) **bicycle_id**: Armazena o id da bicicleta. É obrigatório do tipo UUID e não aceita valor nulo.

2) **color**: Armazena a cor da bicicleta. É um campo obrigatório, do tipo character e não aceita valor nulo.

3) **mark**: Armazena marca da bicicleta. É um campo obrigatório, do tipo character e não aceita valor nulo.

4) **model**: Armazena modelo da bicicleta, adicionando uma restrição na marca. É um campo obrigatório, do tipo character e não aceita valor nulo.

5) **type**: Armazena o tipo da bicicleta. Os valores permitido são, ELETRICA, CARGO E CONVENCIONAL. É um campo obrigatório, do tipo character e não aceita valor nulo.

6) **person_id**: Chave estrageira para a tabela de pessoa. Cada bicicleta está associada a apenas um id de um pessoa.

7) **active**: Armazena uma informação boleana que indica se a bicicleta esta ativa. O sistema permite apenas uma bicicleta ativa por ciclista.

6) **TB_USER**: Tabela resposável por armazenar e gerenciar as informações de usuários. Um usuário define o papel da pessoa no sistema:

1) **user_id**: Armazena o id do usuário. É obrigatório do tipo UUID e não aceita valor nulo.

2) **email**: Armazena o email de cadastro do usuário. É um campo obrigatório, do tipo character e não aceita valor nulo.

3) **password**: Armazena senha de cadastro do usuário. É um campo obrigatório, do tipo character e não aceita valor nulo.

4) **user_type**: Armazena o tipo do usuário, os valores permitidos são, CLIENTE E CICLISTA. É um campo obrigatório, do tipo character e não aceita valor nulo.

7) **TB_RACE_EVALUATION**: Tabela responsável por armazenar e gerenciar as informações referentes às avaliações da corrida. Cliente e ciclista podem realizar avaliações.

1) **evaluation_id**: Armazena o id da avaliação. É obrigatório do tipo UUID e não aceita valor nulo.

2) **race_id**: Armazena o id UUID da corrida. Identifica a corrida pertencente a avaliação. É um campo obrigatório.

3) **evaluator_id**: Armazena o id de quem está realizando a avaliação, pode ser o id do ciclista ou do cliente.

4) **evaluator_type**: Armazena o tipo do usuário que está fazendo a avaliação, os valores permitidos são, CLIENTE E CICLISTA. É um campo obrigatório, do tipo character e não aceita valor nulo.

5) **evaluated_id**: Armazena o id do usário que esta sendo avaliado. Pode ser ciclista ou cliente.

6) **evaluated_type**: Armazena o tipo do usuario que esta sendo avaliado.

7) **score**: Armazena a nota do tipo inteiro que o avaliador vai definir para a corrida. É um campo obrigatório.

8) **description**: Armazena um comentário sobre a corrida. É um campo obrigatório.

9) **date**: Armazena a data e hora que o comentário foi cadastrado. É um campo obrigatório.

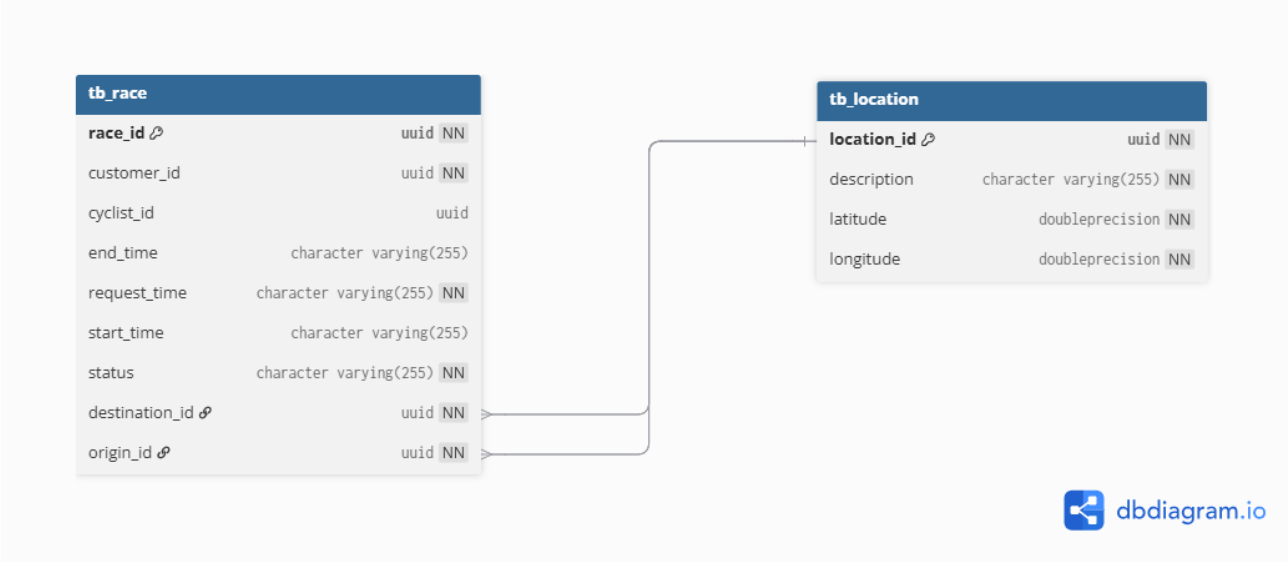


Diagrama físico referente ao banco de dados de corrida.

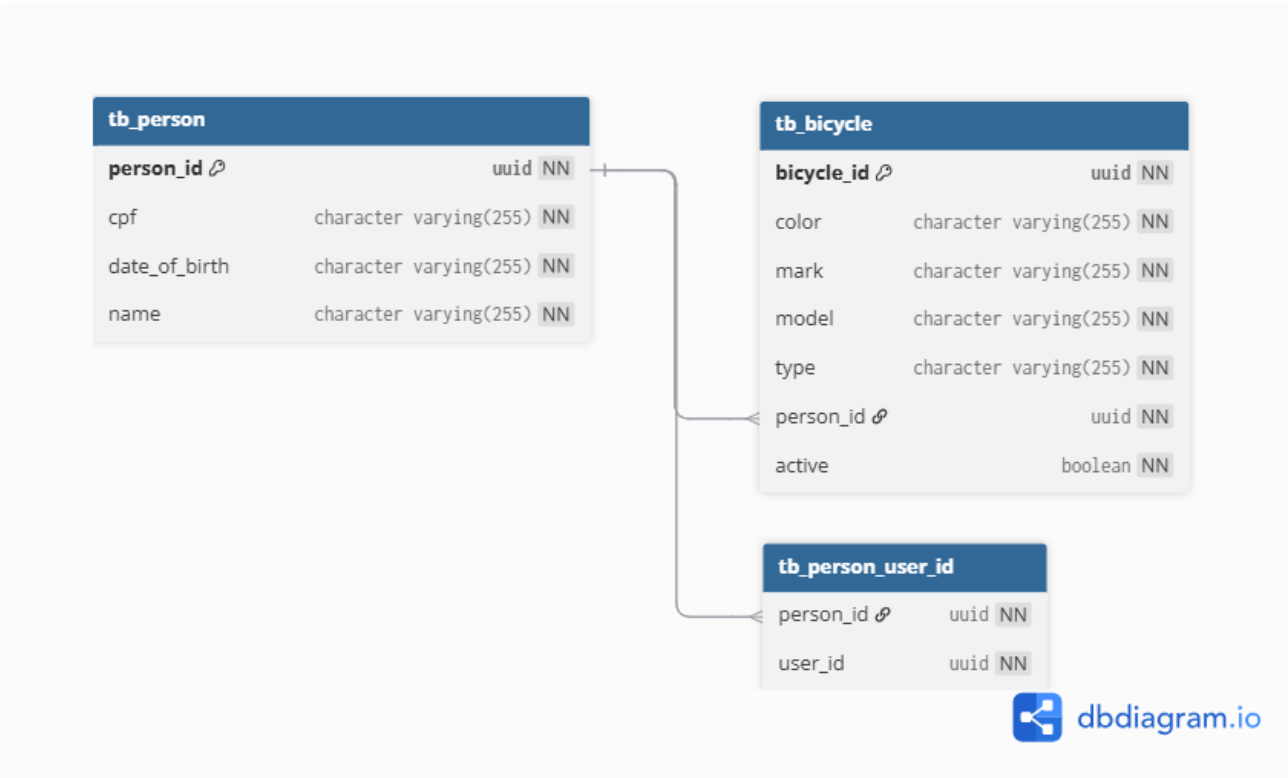


Diagrama físico referente ao banco de dados gerenciamento de pessoas.

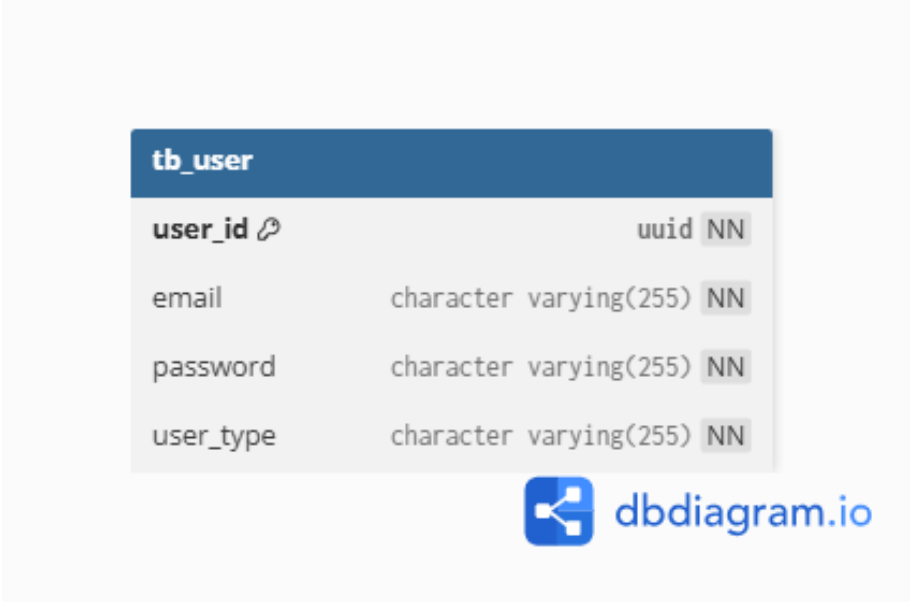


Diagrama físico referente ao banco de dados de autenticação e gerenciamento de usuários.

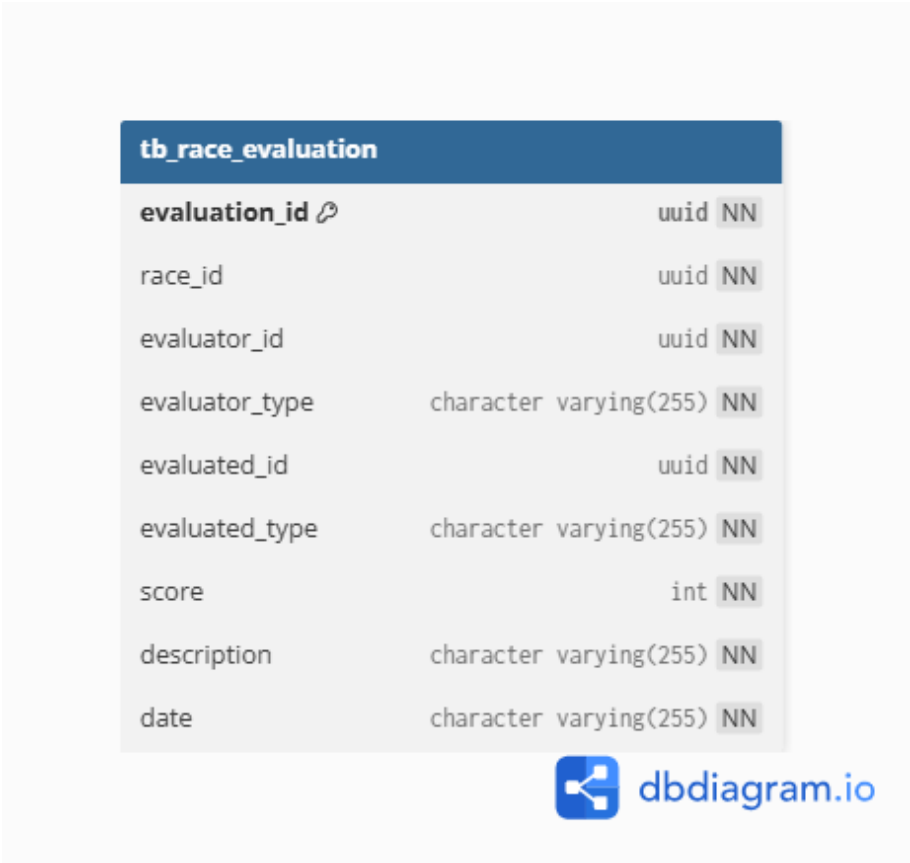


Diagrama físico referente ao banco de dados de avaliação e comentários da corrida.

3.3.3 SQL para criação do banco de dados físico

```
CREATE TABLE tb_location
(
    location_id uuid NOT NULL,
```

```
    description character varying(255) NOT NULL,
    latitude double precision NOT NULL,
    longitude double precision NOT NULL,
    CONSTRAINT location_pkey PRIMARY KEY (location_id)
)
```

Query para criar a tabela TB_LOCATION.

```
CREATE TABLE tb_race
(
    race_id uuid NOT NULL,
    customer_id uuid NOT NULL,
    cyclist_id uuid,
    end_time character varying(255),
    request_time character varying(255) NOT NULL,
    start_time character varying(255),
    status character varying(255) NOT NULL,
    destination_id uuid NOT NULL,
    origin_id uuid NOT NULL,
    CONSTRAINT race_pkey PRIMARY KEY (race_id),
    CONSTRAINT race_origin_fkey FOREIGN KEY (origin_id)
        REFERENCES tb_location(location_id),
    CONSTRAINT race_destination_fkey FOREIGN KEY (destination_id)
        REFERENCES tb_location(location_id)
)
```

Query para criar a tabela TB_RACE.

```
CREATE TABLE tb_person
(
    person_id uuid NOT NULL,
    cpf character varying(255) NOT NULL,
    date_of_birth character varying(255) NOT NULL,
    name character varying(255) NOT NULL,
    CONSTRAINT person_pkey PRIMARY KEY (person_id),
    CONSTRAINT person_cpf_unique UNIQUE (cpf)
)
```

Query para criar a tabela TB_PERSON.

```
CREATE TABLE tb_bicycle
(
    bicycle_id uuid NOT NULL,
    color character varying(255) NOT NULL,
    mark character varying(255) NOT NULL,
    model character varying(255) NOT NULL,
    type character varying(255) NOT NULL,
```

```
    person_id uuid NOT NULL,
    active boolean NOT NULL,
    CONSTRAINT bicycle_pkey PRIMARY KEY (bicycle_id),
    CONSTRAINT bicycle_person_fkey FOREIGN KEY (person_id)
        REFERENCES tb_person (person_id)
)
```

Query para criar a tabela TB_BICYCLE.

```
CREATE TABLE tb_person_user_id
(
    person_id uuid NOT NULL,
    user_id uuid NOT NULL,

    CONSTRAINT person_user_person_fkey FOREIGN KEY (person_id)
        REFERENCES tb_person (person_id)
)
```

Query para criar a tabela TB_PERSON_USER_ID.

```
CREATE TABLE tb_user
(
    user_id uuid NOT NULL,
    email character varying(255) NOT NULL,
    password character varying(255) NOT NULL,
    user_type character varying(255) NOT NULL,
    CONSTRAINT tb_user_pkey PRIMARY KEY (user_id)
)
```

Query para criar a tabela TB_USER.

```
CREATE TABLE tb_race_evaluation
(
    evaluation_id uuid NOT NULL,
    race_id uuid NOT NULL,
    evaluator_id uuid NOT NULL,
    evaluator_type character varying(255) NOT NULL,
    evaluated_id uuid NOT NULL,
    evaluated_type character varying(255) NOT NULL,
    score int NOT NULL,
    description character varying(255) NOT NULL,
    date character varying(255) NOT NULL,
    CONSTRAINT tb_race_evaluation_pkey PRIMARY KEY (evaluation_id)
)
```

Query para criar a tabela TB_RACE_EVALUATION.

4 Qualidade

4.1 Projeto de Testes

Os microsserviços foram testados seguindo dois principais cenários. No primeiro cenário, foi utilizado as ferramentas Postman e Swagger, que permitem realizar requisições http diretamente aos endpoints dos microsserviços. Com isso, foi possível validar o fluxo da requisição desde a entrada, ao cadastro de dados na base de dados e retorno das Apis.

O segundo cenário, foi através de testes de unidade, fundametais para garantir a qualidade e segurança do código. Através dos teste de unidades, é possivel testar o código em pequenas partes, são elas as funções. Com isso, é possivel criar varios cenários de testes para uma função. Quanto maior o numero de testes, com propósitos específicos, maior será a chance de previnir bugs no código ou erros em tempo de execução. Os testes de unidade podem garantir que mudanças futuras no código não quebrem funcionalidades já existentes e também pode ser usado como documentação, facilitando o entendimento do código para outros desenvolvedores.(aws.amazon.com, 2025)

Os testes de unidade foram desenvolvidos utilizando o plugin Kotest para Kotlin. Para simular o comportamento de classes injetadas, classes que são usadas por outras classes, foram utilizados Mocks. Também, todos os testes seguiram o estilo de AnnotationSpec, que é um estilo de teste oferecido pelo plugin Kotest.

4.1.1 Exemplos de teste de unidade

```
class UserControllerTest : AnnotationSpec() {

    private val userUseCase = mockk<UserUseCase>()
    private val userController = UserController(userUseCase)

    private lateinit var user: User
    private lateinit var userRequest: UserRequest
    private val token = "Oidydhegegd.ididdjdkkd.eydydyydydt"

    @BeforeEach
    fun beforeTest() {
        userRequest = UserRequest(
                "usuariodeteste@gmail.com",
                "123456",
                "CLIENTE",
        )

        user = userRequest.toUser()
    }

    @Test
    fun `Deve retornar status code 201 com id do usuario no body`() {
        //cenario
        every { userUseCase.createUser(any()) } answers{ user.userId.toString()
            }
```

```
        //acao
        val result = userController.createUser(userRequest)

        //validacao
        result.statusCode.value() shouldBe HttpStatus.CREATED.value()
        result.body!!.userId shouldBe user.userId.toString()
    }

    @Test
    fun `Deve retornar status code 200 com um token de usuario logado com
        sucesso`() {
        //cenario
        every { userUseCase.login(any()) } answers{ token }

        //acao
        val result = userController.login(UserLoginRequest(user.email, user.
            password))

        //validacao
        result.statusCode.value() shouldBe HttpStatus.OK.value()
        result.body!!.token shouldBe token
    }

    @Test
    fun `Deve lancar excecao quando o login for invalido`() {
        //cenario
        every { userUseCase.login(any()) } answers{ null }

        // acao + validacao
        shouldThrow<UserUnauthorized> {
            userController.login(UserLoginRequest(user.email, user.password))
        }
    }
}
```

Teste de unidade para uma classe de `Controller` validando três cenários de teste.

```
class UserUseCaseImpTest : AnnotationSpec() {

    private val userAdapter = mockk<UserAdapter>()
    private val jwtUtil = mockk<JwtUtil>()
    private val userUseCase = UserUseCaseImp(userAdapter, jwtUtil)

    private lateinit var user: User
    private val token = "Oidydhegegd.ididdjdkkd.eydydyydydt"

    @BeforeEach
    fun beforeTest() {
        user = User(
```

```
            UUID.randomUUID(),
            "usuariodeteste@gmail.com",
            "123456",
            "CLIENTE",
    )
}

@Test
fun `Deve simular a criacao de um usuario retornando o id usuario como
    string`() {
    //cenario
    every { userAdapter.createUser(user) } answers{ user.userId.toString()
        }

    //acao
    val result = userUseCase.createUser(user)

    //validacao
    result shouldBe user.userId.toString()
}

@Test
fun `Deve gerar um token quando os dados de login do usuario estiverem
    corretos`() {
    //cenario
    every { userAdapter.findByEmail(user.email) } answers{ user }
    every { jwtUtil.generateToken(user) } answers{ token }

    //acao
    val result = userUseCase.login(user)

    //validacao
    result shouldBe token
}

@Test
fun `Deve retornar null quando os dados de login do usuario forem invalidos
    `() {
    val invalidUser = User(
            user.userId,
            user.email,
            "7891011",
            "CLIENTE"
    )
    //cenario
    every { userAdapter.findByEmail(any()) } answers{ invalidUser }

    //acao
    val result = userUseCase.login(user)

    //validacao
```

```
        result shouldBe null
    }
}
```

Teste de unidade para uma classe de `Usecase` validando três cenários de teste.

5 Implantação

5.1 Projeto de Implantação

Para implantar a arquitetura de backend com microsserviços, podemos utilizar o ambiente da Amazon AWS. Os microsserviços podem ser implatados no recurso do Amazon Elastic Container Service ECS, onde será necessário gerar uma imagem docker da aplicação através do arquivo .jar dos serviços. Será necessário criar um cluster para cada microsserviço. O cluster executa tasks, que são intancias do microsserviço geradas a partir da imagem docker. Cada microsserviço é independente, podendo ter configurações próprias, assim é possível definir o limite de memória, cpu, número de tarefas e escalabilidade para cada serviço. Para uma menor configuração e mais gerencialmento pela aws o tipo de instancia ECS a ser usada pode ser a Fargate.

Para o serviço de cache, podemos usar o Amazon Elasticache com compatibilidade para o Redis. O amazon Elasticache é um serviço de banco de dados em memória totalmente gerenciado pela AWS, é seguro e oferece alta performance para as aplicações.

Para armazenamento de dados com Postgres podemos usar o Amazon Aurora que é um banco de dados totalmente gerenciado pela AWS, seguro e com alta performance.

Agradecimentos

Gostaria de agradecer primeiramente a Deus, por me conceder a oportunidade de entrar no Instituto Federal da Bahia. Eu sonhei durante muitos anos em realizar um curso na área de tecnologia, e o IFBa me proporcionou a realização desse sonho. Apesar das dificuldades encotradas, eu sempre curtir cada momento do curso, e vejo que evoluí bastante durante esse processo.

Gostaria de agradecer a minha família por ter me dado suporte durante esse processo. O apoio de vocês foi fundamental e sempre será importante em todas as fases da minha vida.

Gostaria de agradecer ao professor Manoel por ter aceitado ser meu orientador e sempre demonstrar estar disponível. Essa característica foi muito importante para que eu conseguisse dar continuidade no desenvolvimento do trabalho de conclusão de curso.

Gostaria de agradecer a todos os professores do curso de Análise e Desenvolvimento de Sistemas do IFBa, todos vocês foram importantes e contribuíram para o meu aprendizado. O Tcc é um resumo de todo o conhecimento que aprendemos durante o curso.

Gostaria de agradecer aos amigos que eu fiz durante o curso, foi muito bom compartilhar essa tragetória com vocês. Compartilhamos muitos momentos juntos, de estudo e descontração, sempre um ajudando o outro, com absoluta certeza vocês contribuíram bastante para que eu chegasse até aqui.

Referências

aws.amazon.com. *O que são testes de unidade*. 2025. Acesso em: 14 jul. 2025. Disponível em: ⟨https://aws.amazon.com/pt/what-is/unit-testing/⟩.

Kotlinlang. *Kotlin*. 2025. Acesso em: 7 jun. 2025. Disponível em: ⟨https://kotlinlang.org/#⟩.

Postgresql.org. *PostgreSql About*. 2025. Acesso em: 7 jun. 2025. Disponível em: ⟨https://www.postgresql.org/about/⟩.

Spring.io. *Spring Boot*. 2025. Acesso em: 7 jun. 2025. Disponível em: ⟨https://spring.io/projects/spring-boot⟩.