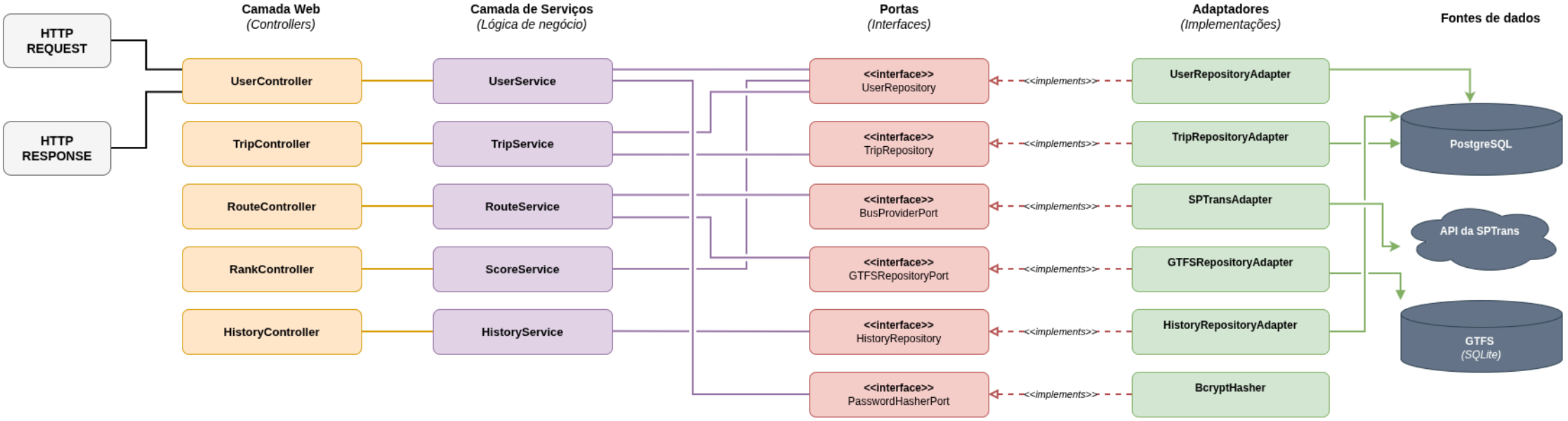
Arquitetura do Backend



Arquitetura do Backend - Bu§P

A arquitetura desenvolvida no projeto utiliza elementos do modelo hexagonal, com portas e adaptadores, para aumentar a facilidade de manutenção, testes e modificações. Além disso, há também elementos do modelo MVC para organizar classes e estruturas em camadas com responsabilidades bem definidas. Ou seja, uma camada de visão que representa o que é mostrado na aplicação, uma de serviços onde fica a lógica de negócio e, por fim, uma camada de banco de dados para armazenamento persistente. Dessa maneira, a aplicação desenvolvida é dividida em:

- Camada Web
- Camada de serviços
- Portas
- Adaptadores
- Fontes de dados

A seguir, é feita uma descrição de cada camada e de suas componentes.

Observações:

Objetos utilizados para armazenar informações são enviados de uma camada a outra como forma de comunicação entre as camadas. Esses objetos pertencem às chamadas classes de domínio que também são descritas abaixo.

GTFS refere-se a uma especificação usada pela SPTrans para troca de informações estáticas do transporte e rotas de ônibus. Em última análise, é um arquivo de rotas de ônibus.

Por causa da grande quantidade de classes, métodos e atributos que compoẽm o projeto, a descrição a seguir prioriza aqueles de maior importância para o entendimento da arquitetura e fluxo de dados. Dessa forma, algumas classes ou métodos foram omitidos ou apenas citados.

-> CLASSES DE DOMÍNIO

User

Responsabilidade: Representa um usuário do sistema.
Atributos: name, email, score, password.
Métodos: nenhum (dataclass).
Transita entre: Adapters, Services e Controllers.

Trip
Responsabilidade: Representa uma viagem registrada.
Atributos: email, route, distance, score, trip_datetime.
Métodos: nenhum (dataclass).
Transita entre: Adapters, Services e Controllers.

BusPosition
Responsabilidade: Representa a posição em tempo real de um ônibus.
Atributos: route_id, latitude, longitude, prefix.
Métodos: nenhum (dataclass).
Transita entre: Adapters (SpTransAdapter), Services (RouteService) e Controllers (RouteController).

BusRoute
Responsabilidade: Representa informações de uma linha de ônibus.
Atributos: route_id, route_name, direction.
Métodos: nenhum (dataclass).
Transita entre: Adapters (SpTransAdapter), Services (RouteService) e Controllers (RouteController).

RouteShape
Responsabilidade: Representa o trajeto geográfico de uma rota.
Atributos: route, shape_id, points.
Métodos: nenhum (dataclass).
Transita entre: Adapters (GTFSRepositoryAdapter), Services (RouteService) e Controllers (RouteController).

UserHistory
Responsabilidade: Representa o histórico de viagens do usuário.
Atributos: user_email, trips.
Métodos: nenhum (dataclass).
Transita entre: Adapters (UserHistoryRepositoryAdapter), Services (HistoryService) e Controllers (HistoryController).

Coordinate
Responsabilidade: Representa um par de coordenadas geográficas.
Atributos: latitude, longitude.
Métodos: nenhum (dataclass).
Transita entre: usada internamente em RouteShape e BusPosition.

-> CAMADA WEB (controllers)

Os Controllers são responsáveis por receber requisições HTTP e delegar o processamento aos services apropriados.

UserController
Responsabilidade: Gerencia endpoints de registro, login e consulta de dados do usuário autenticado.
Usa: UserService.
Atributos: nenhum (dependências injetadas via FastAPI).
Métodos: register, login, get_me.
Endpoints: POST /register, POST /login, GET /me.
Recebe: User da camada de Services (UserService).
Envia: User para o cliente (via JSON/schema de resposta).

TripController
Responsabilidade: Gerencia o endpoint de criação de viagens.
Usa: TripService.
Atributos: nenhum (dependências injetadas via FastAPI).
Métodos: create_trip.
Endpoints: POST /trips/.
Recebe: Trip da camada de Services (TripService).
Envia: Trip para o cliente (via JSON/schema de resposta).

RouteController
Responsabilidade: Gerencia endpoints de busca de rotas, posições em tempo real e shapes geográficos.
Usa: RouteService.
Atributos: nenhum (dependências injetadas via FastAPI).
Métodos: search_routes_endpoint, get_positions_endpoint, get_shapes_endpoint.
Endpoints: GET /search, GET /positions, GET /shapes.
Recebe: BusPosition, BusRoute e RouteShape da camada de Services (RouteService).
Envia: BusPosition, BusRoute e RouteShape para o cliente (via JSON/schema de resposta).

RankController
Responsabilidade: Gerencia endpoints de ranking individual e global.
Usa: ScoreService.
Atributos: nenhum (dependências injetadas via FastAPI).
Métodos: get_user_ranking, get_global_ranking.
Endpoints: GET /user, GET /global.
Recebe: User (com score) da camada de Services (ScoreService).
Envia: dados de ranking para o cliente (via JSON/schema de resposta).

HistoryController
Responsabilidade: Gerencia o endpoint de consulta de histórico de viagens.
Usa: HistoryService.
Atributos: nenhum (dependências injetadas via FastAPI).
Métodos: get_user_history.
Endpoints: GET /history/.
Recebe: UserHistory da camada de Services (HistoryService).
Envia: UserHistory para o cliente (via JSON/schema de resposta).

-> CAMADA DE SERVIÇOS (Services / Lógica de Negócio)

Seguindo o modelo de arquitetura hexagonal, os services representam a lógica de negócio da aplicação, isto é, representam o core dela. Ademais, cada service depende de uma ou mais interfaces (portas) para acessar dados ou funcionalidades externas. As implementações concretas são injetadas em tempo de execução seguindo a lógica de inversão de dependência própria da arquitetura hexagonal.

UserService
Responsabilidade: Gerencia a criação de usuários, autenticação e consulta de dados de usuários. Utiliza o UserRepository para persistência e o PasswordHasherPort para hash e verificação de senhas.
Usa: UserRepository, PasswordHasherPort.
Atributos: user_repository, password_hasher.
Métodos: create_user, get_user, login_user.
Envia: User para a camada Web (UserController).

TripService
Responsabilidade: Gerencia o registro de viagens dos usuários. Utiliza o TripRepository para salvar viagens e o UserRepository para atualizar a pontuação do usuário após cada viagem.
Usa: TripRepository, UserRepository.
Atributos: trip_repository, user_repository.
Métodos: create_trip.
Envia: Trip para a camada Web (TripController).

RouteService
Responsabilidade: Fornece informações sobre rotas de ônibus. Utiliza o BusProviderPort para obter posições em tempo real e buscar rotas, e o GTFSRepositoryPort para obter o trajeto geográfico das rotas.
Usa: BusProviderPort, GTFSRepositoryPort.
Atributos: bus_provider, gtfs_repository.
Métodos: get_bus_positions, search_routes, get_route_shapes.
Envia: BusPosition, BusRoute e RouteShape para a camada Web (RouteController).

ScoreService
Responsabilidade: Gerencia o sistema de ranking. Consulta o UserRepository para obter a pontuação individual do usuário e o ranking global ordenado.
Usa: UserRepository.
Atributos: user_repository.
Métodos: get_user_ranking, get_global_ranking.
Envia: User (com score) para a camada Web (RankController).

HistoryService
Responsabilidade: Fornece o histórico de viagens de um usuário. Consulta o HistoryRepository para obter todas as viagens registradas.
Usa: HistoryRepository.
Atributos: history_repository.
Métodos: get_user_history.
Envia: UserHistory para a camada Web (HistoryController).

-> CAMADA DE PORTAS (Ports / Interfaces)

As portas são interfaces abstratas que definem contratos para acesso a dados e funcionalidades externas. Os services dependem dessas interfaces, não de implementações concretas, seguindo a lógica de uma arquitetura hexagonal.

UserRepository
Responsabilidade: Define o contrato para persistência e consulta de usuários.
Atributos: nenhum (interface abstrata).
Métodos: save_user, get_user_by_email, get_all_users_ordered_by_score, add_user_score.
Recebe: User da camada de Services (UserService, TripService, ScoreService).
Retorna: User para a camada de Services.

TripRepository
Responsabilidade: Define o contrato para persistência de viagens.
Atributos: nenhum (interface abstrata).
Métodos: save_trip.
Recebe: Trip da camada de Services (TripService).

BusProviderPort
Responsabilidade: Define o contrato para obtenção de dados de ônibus em tempo real.
Atributos: nenhum (interface abstrata).
Métodos: get_bus_positions, search_routes.
Retorna: BusPosition e BusRoute para a camada de Services (RouteService).

GTFSRepositoryPort
Responsabilidade: Define o contrato para consulta de dados estáticos GTFS.
Atributos: nenhum (interface abstrata).
Métodos: get_route_shape.
Retorna: RouteShape para a camada de Services (RouteService).

HistoryRepository
Responsabilidade: Define o contrato para consulta de histórico de viagens.
Atributos: nenhum (interface abstrata).
Métodos: get_user_history.
Retorna: UserHistory para a camada de Services (HistoryService).

PasswordHasherPort
Responsabilidade: Define o contrato para hash e verificação de senhas.
Atributos: nenhum (interface abstrata).
Métodos: hash, verify.
Retorna: string (hash) para a camada de Services (UserService).

-> CAMADA DE ADAPTADORES (Adapters / Implementações)

Os Adaptadores são implementações concretas das portas. Cada adaptador se conecta a uma fonte de dados específica para realizar as operações definidas na interface.

UserRepositoryAdapter
Responsabilidade: Persiste e recupera dados de usuários no banco de dados relacional PostgreSQL.
Implementa: UserRepository.
Conecta-se a: PostgreSQL (via SQLAlchemy).
Atributos: session.
Métodos: save_user, get_user_by_email, get_all_users_ordered_by_score, add_user_score.
Recebe: User da camada de Services (via Port).
Retorna: User para a camada de Services (via Port).

TripRepositoryAdapter
Responsabilidade: Persiste dados de viagens no banco de dados PostgreSQL.
Implementa: TripRepository.
Conecta-se a: PostgreSQL (via SQLAlchemy).
Atributos: session.
Métodos: save_trip.
Recebe: Trip da camada de Services (via Port).

UserHistoryRepositoryAdapter
Responsabilidade: Recupera o histórico completo de viagens de um usuário, incluindo relacionamentos com a tabela de viagens.
Implementa: HistoryRepository.
Conecta-se a: PostgreSQL (via SQLAlchemy).
Atributos: session.
Métodos: get_user_history.
Retorna: UserHistory para a camada de Services (via Port).

SpTransAdapter
Responsabilidade: Comunica-se com a API da SPTrans para obter posições de ônibus em tempo real e realizar buscas de linhas.
Implementa: BusProviderPort.
Conecta-se a: API SPTrans (HTTP).
Atributos: api_token, base_url, client.
Métodos: get_bus_positions, search_routes.
Retorna: BusPosition e BusRoute para a camada de Services (via Port).

GTFSRepositoryAdapter
Responsabilidade: Consulta o banco de dados GTFS (formato SQLite) para obter dados estáticos de rotas, como o shape geográfico dos trajetos.
Implementa: GTFSRepositoryPort.
Conecta-se a: GTFS SQLite.
Atributos: nenhum (usa conexão via função get_gtfs_db).
Métodos: get_route_shape.
Retorna: RouteShape para a camada de Services (via Port).

PasslibPasswordHasher
Responsabilidade: Realiza hash de senhas usando bcrypt_sha256 e verifica senhas em operações de autenticação.
Implementa: PasswordHasherPort.
Conecta-se a: Biblioteca Passlib/Bcrypt (em memória).
Atributos: nenhum (usa contexto global _pwd_ctx).
Métodos: hash, verify.
Retorna: string (hash) e boolean (verificação) para a camada de Services (via Port).

-> FONTES DE DADOS

As fontes de dados são os sistemas externos onde os dados são efetivamente armazenados ou obtidos. Os Adapters são responsáveis por se comunicar com essas fontes, traduzindo operações de domínio para operações específicas de cada tecnologia.

PostgreSQL (Banco de Dados Principal)

Descrição: Banco de dados relacional utilizado como persistência principal da aplicação. Armazena todos os dados de usuários, viagens e histórico de forma estruturada e com suporte a transações ACID.
Dados armazenados: Usuários (nome, email, senha hash, pontuação), Viagens (rota, distância, pontuação, data/hora).
Tecnologia de acesso: SQLAlchemy (ORM assíncrono).
Adapters que utilizam: UserRepositoryAdapter, TripRepositoryAdapter, UserHistoryRepositoryAdapter.

API SPTrans (Serviço Externo)

Descrição: API REST externa fornecida pela São Paulo Transporte S.A. que disponibiliza dados em tempo real sobre o transporte público da cidade de São Paulo. Requer autenticação via token.
Dados fornecidos: Posições em tempo real dos ônibus (latitude, longitude, prefixo do veículo), informações de linhas (código, nome, sentido).
Tecnologia de acesso: Cliente HTTP assíncrono (httpx).
Adapters que utilizam: SpTransAdapter.

GTFS SQLite (Banco de Dados Estático)

Descrição: Banco de dados SQLite local contendo dados estáticos de transporte público no formato GTFS (General Transit Feed Specification). O GTFS é um padrão aberto desenvolvido pelo Google para compartilhamento de informações de transporte público.
Dados armazenados: Shapes das rotas (sequência de coordenadas que formam o trajeto no mapa), informações de trips (viagens programadas), rotas e direções.
Tecnologia de acesso: SQLite3 nativo do Python com connection pooling.
Adapters que utilizam: GTFSRepositoryAdapter.

-> FLUXO RESUMIDO

O fluxo de dados na arquitetura segue, em geral, o seguinte padrão:

Ida (Requisição):
Cliente -> Requisição HTTP -> Controller -> Service -> Port (interface) -> Adapter -> Fonte de Dados

Volta (Resposta):
Fonte de Dados -> Adapter -> Port (interface) -> Service -> Controller -> Resposta HTTP -> Cliente

Exemplo de fluxo de login (ida e volta):

Ida:
1. Cliente envia requisição POST /login com email e senha no corpo JSON.
2. UserController.login() recebe a requisição e extrai os dados do schema LoginRequest.
3. UserController chama UserService.login_user(email, password).
4. UserService chama UserRepository.get_user_by_email(email) para buscar o usuário.
5. UserRepositoryAdapter (implementação do Port) executa query SQL no PostgreSQL.
6. PostgreSQL retorna os dados do usuário (ou nenhum registro).

Volta:
7. UserRepositoryAdapter converte o registro do banco (UserDB) para o objeto de domínio User.
8. User é retornado para o UserService através do Port UserRepository.
9. UserService chama PasswordHasherPort.verify(password, user.password) para validar a senha.
10. PasslibPasswordHasher compara o hash usando bcrypt e retorna True ou False.
11. Se válido, UserService retorna o objeto User para o UserController.
12. UserController converte User para o schema de resposta (UserResponse) e gera o token JWT.
13. Cliente recebe resposta HTTP 200 com o token e dados do usuário em JSON.

Exemplo de fluxo de obter posições de ônibus (ida e volta):

Ida:
1. Cliente envia requisição GET /positions?route_id=1234.
2. RouteController.get_positions_endpoint() recebe a requisição e extrai o route_id.
3. RouteController chama RouteService.get_bus_positions(route_id).
4. RouteService chama BusProviderPort.get_bus_positions(route_id).
5. SpTransAdapter (implementação do Port) faz requisição HTTP GET à API SPTrans.
6. API SPTrans retorna JSON com posições dos ônibus da linha.

Volta:
7. SpTransAdapter converte a resposta JSON (SPTransPositionsResponse) para lista de BusPosition.
8. Lista de BusPosition é retornada para o RouteService através do Port BusProviderPort.
9. RouteService retorna a lista de BusPosition para o RouteController.
10. RouteController converte cada BusPosition para o schema de resposta (BusPositionResponse).
11. Cliente recebe resposta HTTP 200 com lista de posições em JSON.

Exemplo de fluxo de obter shapes de rotas (ida e volta):

Ida:
1. Cliente envia requisição GET /shapes?bus_line=8000&direction=1.
2. RouteController.get_shapes_endpoint() recebe a requisição e extrai os parâmetros.
3. RouteController chama RouteService.get_route_shapes(bus_line, direction).
4. RouteService chama GTFSRepositoryPort.get_route_shape(route_identifier).
5. GTFSRepositoryAdapter executa query SQL no banco GTFS SQLite.
6. SQLite retorna os registros de shapes (coordenadas ordenadas por sequência).

Volta:
7. GTFSRepositoryAdapter converte os registros para objeto RouteShape com lista de RouteShapePoint.
8. RouteShape é retornado para o RouteService através do Port GTFSRepositoryPort.
9. RouteService retorna RouteShape para o RouteController.
10. RouteController converte RouteShape para o schema de resposta (RouteShapeResponse).
11. Cliente recebe resposta HTTP 200 com o trajeto geográfico em JSON.

Exemplo de fluxo de criar viagem (ida e volta):

Ida:
1. Cliente envia requisição POST /trips/ com dados da viagem (rota, distância) e token JWT.
2. TripController.create_trip() recebe a requisição, valida o token e extrai os dados.
3. TripController chama TripService.create_trip(email, route, distance).
4. TripService calcula a pontuação baseada na distância.
5. TripService chama TripRepository.save_trip(trip) para persistir a viagem.
6. TripRepositoryAdapter executa INSERT no PostgreSQL.
7. TripService chama UserRepository.add_user_score(email, score) para atualizar pontuação.
8. UserRepositoryAdapter executa UPDATE no PostgreSQL.

Volta:
9. PostgreSQL confirma as operações.
10. UserRepositoryAdapter retorna confirmação para o TripService.
11. TripRepositoryAdapter retorna o objeto Trip salvo para o TripService.
12. TripService retorna Trip para o TripController.
13. TripController converte Trip para o schema de resposta (TripResponse).
14. Cliente recebe resposta HTTP 201 com os dados da viagem criada em JSON.

-> DIAGRAMA DO BANCO DE DADOS

Apesar da aplicação ter um diagrama de banco de dados simples, a modelagem dessa parte do sistema foi bastante importante para o planejamento do restante da arquitetura. A partir dela, foi possível entender como os dados seriam armazenados e de que forma

teríamos que construir repositórios para interagir com essas informações. Além disso, foi essencial para o entendimento de quais dados seriam responsabilidade da API da SPTrans e quais dados seriam armazenados pelo backend.