Documentação Backend

- WeaveTrip -

Diego Silva Oliveira

Edivan Figueiredo Braga

Eduardo Caetano de Souza Junior

Fernando Silva Ferreira

Franklin Xavier Ramos Santos

Geisa de Souza Santos

Marcus Vítor Souza Cardoso

Marcos Morais de Sousa

SUMÁRIO

1. VISÃO GERAL DO DOCUMENTO

Este documento tem como principal objetivo apresentar, de forma clara e detalhada, a estrutura, funcionalidades e regras de negócio da API desenvolvida para o sistema WeaveTrip. A finalidade da documentação é servir como uma fonte de referência técnica para todos os envolvidos no desenvolvimento, manutenção, testes e integração da aplicação, garantindo o alinhamento entre as equipes e facilitando futuras expansões ou correções.

A documentação descreve os principais módulos do backend, incluindo entidades como usuários, reservas, pacotes de viagem e mídias associadas, abordando desde os fluxos básicos de criação e consulta até as regras específicas de validação, permissões de acesso e controle de integridade. Também são cobertos aspectos importantes como autenticação, controle de sessões, estrutura de banco de dados e rotas de acesso (REST e GraphQL).

O escopo deste material está restrito ao backend do sistema WeaveTrip, não abrangendo componentes de frontend web ou mobile. Essa delimitação garante que a documentação seja focada nos aspectos técnicos e operacionais do servidor da aplicação, promovendo uma compreensão aprofundada das funcionalidades expostas pela API e sua lógica de funcionamento.

Público-alvo: desenvolvedores backend, QA, DevOps, stakeholders técnicos.

2. OBJETIVO DO SISTEMA

O WeaveTrip é uma plataforma de gestão de pacotes turísticos desenvolvida para facilitar o controle e a organização de reservas, usuários e ofertas de viagens por meio de uma arquitetura backend robusta e escalável. O sistema permite que agências de turismo cadastrem, atualizem e gerenciem pacotes de viagem, incluindo o controle de vagas disponíveis, políticas de cancelamento, formas de pagamento e o processo completo de reservas pelos usuários finais. Focado em automação e eficiência operacional, o WeaveTrip visa centralizar funcionalidades essenciais para o planejamento e administração de viagens em um único sistema, promovendo maior controle, segurança e agilidade para clientes, agências e administradores da plataforma.

Funcionalmente, o sistema permite que clientes visualizem pacotes de viagem cadastrados por agências, realizem reservas conforme disponibilidade de

vagas, acompanhem o status de suas reservas e realizem pagamentos conforme as opções oferecidas. O WeaveTrip também oferece funcionalidades administrativas para agências e administradores, como cadastro e edição de pacotes, definição de políticas de cancelamento, controle de quantidade de vagas disponíveis e gestão de usuários. Toda a experiência é voltada para simplificar o processo de oferta, reserva e controle de viagens dentro de um ambiente digital centralizado. Além disso, a plataforma centraliza e automatiza etapas importantes do planejamento de viagens, como reserva de serviços, aplicação de políticas comerciais e controle de disponibilidade de pacotes.

O WeaveTrip busca solucionar um problema recorrente no setor de turismo: a fragmentação dos serviços e a dificuldade em conciliar as preferências dos viajantes com as opções disponíveis no mercado. Por meio da integração inteligente de recursos e da centralização das operações em um backend robusto — desenvolvido com Elixir, Phoenix, PostgreSQL e GraphQL — o sistema garante segurança, escalabilidade e eficiência para aplicações web e mobile conectadas.

A proposta de valor do WeaveTrip reside em sua capacidade de transformar o planejamento de viagens em uma jornada fluida, centralizada e inteligente. Ao fornecer recomendações alinhadas com os desejos do usuário e automatizar tarefas operacionais como controle de vagas, confirmações e cancelamentos de reservas, a plataforma agrega conveniência, personalização e eficiência ao setor. Comparado a soluções concorrentes como TripAdvisor, Airbnb e Booking.com, o WeaveTrip se diferencia pelo uso estratégico de inteligência artificial para recomendações personalizadas e pela integração de múltiplos serviços turísticos em um único ecossistema.

Por fim, o projeto é gerenciado com o suporte de ferramentas como AirTable — que alia a simplicidade de uma planilha à estrutura de banco de dados — e FAZ, que facilita a gestão de processos e promove um fluxo de trabalho mais eficiente. Essas ferramentas contribuem diretamente para a organização interna e para o desenvolvimento ágil e colaborativo da plataforma.

Público-alvo:

- Clientes interessados em viagens personalizadas.
- Agências de turismo que criam, oferecem e gerenciam pacotes.

- Administradores da plataforma com funções de controle e auditoria.
- Viajantes de lazer, aventureiros, profissionais em viagens de negócios e turistas de luxo, cada um com perfis e expectativas distintas atendidas de forma segmentada.

3. TECNOLOGIAS E FERRAMENTAS UTILIZADAS

Nesta seção, são detalhadas as principais tecnologias e ferramentas utilizadas no desenvolvimento do backend do WeaveTrip. A seguir, abordaremos os seguintes pontos: (3.1) linguagens de programação e suas versões, explicando os motivos que fundamentam sua escolha para o projeto; (3.2) frameworks principais que sustentam a arquitetura do sistema, destacando suas funções específicas; (3.3) bibliotecas e dependências adicionais que complementam a funcionalidade do backend, com ênfase em segurança, validação e integração; e (3.4) ferramentas empregadas para garantir a qualidade do código e a robustez dos testes automatizados, fundamentais para a manutenção e evolução do sistema.

3.1 Linguagens de Programação e Versões

O backend do WeaveTrip é desenvolvido principalmente em Elixir 1.18.2, compilado com Erlang/OTP 27, o que garante alta performance, escalabilidade e concorrência eficiente, características essenciais para sistemas web modernos. Elixir é conhecido por sua robustez no desenvolvimento de aplicações distribuídas e fault-tolerant, além de possuir uma comunidade ativa e crescente, que contribui com diversas bibliotecas para acelerar o desenvolvimento.

Para o desenvolvimento do backend, utiliza-se o framework Phoenix 1.7.20, que oferece uma estrutura leve, rápida e escalável para a construção de APIs e aplicações web. O Phoenix é amplamente adotado por sua integração nativa com Elixir, suporte a WebSockets, canais e um ecossistema rico para desenvolvimento backend. O sistema também emprega o GraphQL por meio das bibliotecas Absinthe (versão ~> 1.7) e seus complementos para Phoenix, o que permite uma API flexível e eficiente para consumo frontend.

3.2 Frameworks Principais

O Phoenix Framework (versão 1.7.20) é o principal framework web utilizado no projeto. Ele oferece uma base robusta, escalável e de alta performance para o desenvolvimento de aplicações web e APIs. Com sua arquitetura orientada a processos, o Phoenix permite gerenciar conexões simultâneas de forma eficiente, tornando-o ideal para sistemas que demandam alta concorrência, como plataformas de reserva e gestão de viagens. Além disso, o Phoenix facilita a construção de APIs REST e a integração com WebSockets para comunicação em tempo real, características que podem ser exploradas para melhorar a experiência do usuário.

O Absinthe (versão ~> 1.7) é o framework escolhido para implementação da API GraphQL no backend. Ele possibilita a criação de APIs flexíveis e eficientes, permitindo que o frontend consuma dados de forma otimizada, solicitando exatamente as informações necessárias em uma única requisição. Absinthe integra-se nativamente com Phoenix, garantindo que a comunicação entre cliente e servidor seja segura e performática. Esse framework facilita o desenvolvimento de queries, mutations e subscriptions, ampliando a capacidade da API para suportar funcionalidades dinâmicas e interativas.

O MinIO é o serviço adotado para armazenamento e gerenciamento de arquivos de mídia no backend do WeaveTrip, como fotos e vídeos associados aos pacotes de viagem. Compatível com a API S3 da Amazon Web Services, o MinIO permite a integração com bibliotecas como ExAws, ExAws.S3, hackney e sweet_xml, viabilizando operações seguras de upload, download e acesso a objetos diretamente pela aplicação. Essa solução oferece alta performance, confiabilidade e controle local sobre os dados armazenados, eliminando a dependência de serviços externos e garantindo maior flexibilidade na infraestrutura de armazenamento. O uso do MinIO é essencial para manter a escalabilidade e integridade dos recursos multimídia do sistema.

Em conjunto, Phoenix e Absinthe proporcionam uma infraestrutura moderna e eficiente que atende aos requisitos do WeaveTrip, garantindo desempenho, escalabilidade e flexibilidade na construção da API backend. Já o MinIO complementa essa arquitetura ao oferecer uma solução leve e compatível com o S3 para o armazenamento e gerenciamento de arquivos de mídia, como fotos e vídeos

associados aos pacotes de viagem, assegurando alta disponibilidade e integração simplificada com o sistema.

3.3 Bibliotecas e Dependências Adicionais

1. **Guardian (~> 2.4)**
   Responsável pela autenticação via JWT (JSON Web Token), permitindo o controle seguro de sessões e acesso a recursos protegidos da API.

2. **Joken (~> 2.6)**
   Biblioteca complementar usada internamente pelo Guardian para a codificação, decodificação e verificação de tokens JWT.

3. **Pbkdf2Elixir (~> 2.1)**
   Utilizada para a criptografia segura de senhas, aplicando o algoritmo PBKDF2 para gerar hashes resistentes a ataques de força bruta.

4. **Brcpfcnpj (~> 2.1)**
   Realiza a validação de documentos brasileiros, como CPF e CNPJ, garantindo a integridade dos dados cadastrais de usuários e empresas.

5. **CorsPlug (~> 3.1)**
   Habilita e configura o CORS (Cross-Origin Resource Sharing), permitindo que o frontend acesse a API mesmo quando hospedado em um domínio diferente.

6. **ExAws (~> 2.5)**
   Biblioteca base para integração com serviços da AWS, utilizada no projeto para comunicação com serviços compatíveis com S3.

7. **ExAws.S3 (~> 2.5)**
   Módulo específico do ExAws para operações de upload, download e manipulação de arquivos em buckets S3 — neste caso, utilizado com o MinIO.

8. **Hackney (~> 1.20)**
   Cliente HTTP de baixo nível utilizado internamente pelo ExAws para realizar requisições HTTP assíncronas de forma eficiente.

9. **SweetXml (~> 0.7.4)**
   Usada para parsear e manipular respostas XML retornadas por serviços como o S3, quando necessário.

10. **Mime (~> 2.0)**
    Garante o correto mapeamento e tratamento de tipos MIME (tipos de mídia), importante para upload e download de arquivos com tipos específicos.

11. **Swoosh (~> 1.18)**
    Responsável pelo envio de e-mails na aplicação, como confirmações de cadastro, notificações ou recuperação de senha.

12. **GenSMTP (~> 1.1.1)**
    Utilizada em conjunto com o Swoosh para envio de e-mails via protocolo SMTP, oferecendo suporte a servidores de e-mail personalizados.

13. **Jason (~> 1.4)**
    Biblioteca JSON utilizada para serialização e desserialização de dados, essencial na comunicação entre API e clientes GraphQL/REST.

14. **TelemetryMetrics (~> 1.0)** e **TelemetryPoller (~> 1.0)**
    Empregadas para coleta e análise de métricas internas da aplicação, úteis para monitoramento de performance e uso de recursos.

15. **PhoenixLiveDashboard (~> 0.8.3)**
    Interface interativa para visualização de métricas e estado da aplicação em tempo real, utilizada durante o desenvolvimento e manutenção.

16. **DNSCluster (~> 0.1.1)**
    Auxilia na formação de clusters de nós Elixir por meio de resolução DNS, útil para ambientes distribuídos.

17. **Phoenix (~> 1.7.20)**
    Framework web completo que provê ferramentas para criação de aplicações escaláveis, performáticas e reativas.

18. **PhoenixEcto (~> 4.5)**
    Integração entre Phoenix e Ecto, permitindo que recursos como formulários, erros e inserções se comuniquem com o banco de dados.

19. **EctoSQL (~> 3.10)**
    Extensão do Ecto para suporte a SQL e bancos relacionais como PostgreSQL, incluindo suporte a migrations, consultas e conexões.

20. **Postgrex (>= 0.0.0)**
    Driver PostgreSQL nativo em Elixir, utilizado para conectar e executar comandos no banco de dados.

21. **Gettext (~> 0.26)**
    Responsável pela internacionalização (i18n) da aplicação, permitindo tradução de textos de forma organizada.

22. **Bandit (~> 1.5)**
    Servidor HTTP nativo em Elixir, leve e altamente integrável, usado como alternativa ao Cowboy.

23. **Absinthe (> 1.7), AbsinthePlug (> 1.5), AbsinthePhoenix (~> 2.0)**
    Conjunto de bibliotecas que implementam suporte completo a GraphQL, possibilitando criação de APIs flexíveis, eficientes e compatíveis com frontends modernos.

24. **Dotenv (~> 3.0.0)**
    Utilitário que carrega variáveis de ambiente a partir de arquivos .env, facilitando o gerenciamento de configurações sensíveis em diferentes ambientes.

25. **StripityStripe (~> 3.2.0)**
    Biblioteca de integração com a API da Stripe para realização de pagamentos, gerenciamento de clientes e cobranças.

3.4 Ferramentas de Testes e Qualidade de Código

O Elixir é uma linguagem funcional moderna construída sobre a máquina virtual do Erlang (BEAM), voltada à criação de sistemas concorrentes, distribuídos e tolerantes a falhas. Um dos recursos mais relevantes da linguagem é seu suporte nativo a testes, por meio do framework ExUnit, que fornece um ambiente robusto para validação do comportamento de software.

O ExUnit é o framework de testes integrado da linguagem e contempla todas as ferramentas necessárias para a construção e execução de testes automatizados. Os testes são implementados como scripts Elixir, sendo, portanto, salvos com a extensão .exs. Antes da execução dos testes, é necessário inicializar o ExUnit por meio do comando ExUnit.start(), procedimento usualmente realizado no arquivo test/test_helper.exs (PROGPOINTER, 2023).

Ao gerar um novo projeto com mix, um teste de exemplo é automaticamente criado e pode ser encontrado em test/example_test.exs, servindo como ponto de partida para a estruturação da suíte de testes.

Testes Unitários

Os testes unitários têm como objetivo verificar pequenas partes do código de forma isolada, como funções ou módulos específicos. Essa abordagem permite assegurar que cada unidade do sistema se comporta como esperado. Além de possibilitar a identificação precoce de falhas, os testes unitários promovem confiança durante refatorações e servem como documentação executável (FOWLER, 2022).

As principais características dos testes unitários são: (i) rapidez, por serem executados em milissegundos; (ii) isolamento, pois não dependem de recursos externos como bancos de dados ou APIs; e (iii) determinismo, o que garante que produzam os mesmos resultados para os mesmos inputs.

Testes de Integração

Os testes de integração são responsáveis por verificar a interação entre diferentes módulos, sistemas ou serviços. Eles são aplicados para identificar falhas que não emergem em testes unitários, como incompatibilidades na troca de dados, erros de comunicação entre componentes ou comportamentos inesperados quando múltiplos elementos trabalham em conjunto.

Entre seus objetivos destacam-se: validar a interoperabilidade entre componentes; identificar chamadas incorretas; garantir a fluidez no fluxo de dados; e certificar que os contratos e interfaces estejam sendo respeitados.

Três abordagens principais são adotadas para testes de integração:

- Big Bang: todos os módulos são integrados simultaneamente. É rápido, mas dificulta a detecção de falhas específicas.
- Incremental (Top-Down ou Bottom-Up): integração e teste são feitos gradualmente, reduzindo riscos e facilitando o diagnóstico de problemas.
- Sanduíche (Híbrido): combinação dos métodos anteriores para balancear cobertura e eficiência.

No contexto do Elixir, os testes de integração são amplamente utilizados para validar a comunicação entre módulos, interações com bancos de dados, chamadas HTTP e sistemas distribuídos. Ferramentas como Mox e Mock facilitam a criação de mocks e stubs, enquanto o ExUnit continua sendo a base da execução dos testes.

Testes de Documentação (Doctests)

Os doctests representam uma abordagem eficaz para unir documentação e validação de código. Por meio da inclusão de exemplos executáveis em comentários de documentação (com uso de @moduledoc e @doc), é possível garantir que a documentação reflita com precisão o comportamento real do sistema.

Blocos iniciados com iex> são executados automaticamente durante a rotina mix test, e seus resultados são comparados com as saídas esperadas. Caso ocorra divergência, o teste é sinalizado como falho.

Entre os benefícios dessa técnica, destacam-se: atualização automática da documentação, eliminação de exemplos desatualizados, validação contínua de especificações, e melhoria na curva de aprendizado para novos desenvolvedores (VALIM, 2022). Para que os doctests sejam efetivos, recomenda-se focar em exemplos simples, ilustrativos e significativos, utilizando testes unitários tradicionais para casos mais complexos.

Testes de Propriedade (Property-Based Testing)

Os testes de propriedade, ou property-based testing, constituem uma alternativa aos testes baseados em exemplos específicos. Em vez de verificar entradas e saídas pontuais, define-se propriedades invariantes que o sistema deve satisfazer em qualquer cenário.

O framework gera automaticamente centenas de entradas aleatórias para verificar se as propriedades continuam válidas, promovendo uma cobertura de testes mais ampla e robusta. Em Elixir, bibliotecas como StreamData viabilizam essa abordagem.

Testes de Carga e Estresse

Embora não estejam integrados diretamente ao ExUnit, os testes de carga e estresse são fundamentais para sistemas Elixir, dada a natureza concorrente e altamente distribuída da linguagem. Ferramentas como Benchee são utilizadas para realizar benchmarks de desempenho, mensurando tempos de execução, throughput e comportamento sob estresse.

Testes End-to-End (E2E)

Os testes end-to-end (E2E) são projetados para simular o uso real do sistema do ponto de vista do usuário final. Eles validam não apenas funcionalidades isoladas, mas toda a cadeia de operação — da interface à persistência de dados.

No ecossistema Elixir, algumas ferramentas se destacam:

- Wallaby: framework robusto para automação de navegadores com sintaxe expressiva.
- Hound: permite controle programático de navegadores.
- Bypass: possibilita o mock de serviços externos durante os testes E2E.

- ExUnit: coordena e executa os testes, sendo o núcleo da suíte de testes Elixir.

Essas ferramentas, combinadas, viabilizam uma arquitetura de testes capaz de abranger desde a verificação de funções isoladas até a simulação de fluxos completos de interação, permitindo uma cobertura de testes ampla, coesa e eficaz.

4. REFERÊNCIAS E FONTES DE CONSULTA

Durante o desenvolvimento do backend do projeto WeaveTrip, foram consultadas diversas fontes técnicas que serviram como base para a implementação das funcionalidades, definição da arquitetura e organização do código. A documentação oficial do framework Phoenix e da linguagem Elixir foi fundamental para compreender a estrutura do projeto, o ciclo de vida das requisições e o funcionamento interno das ferramentas (PHOENIX FRAMEWORK, 2023; ELIXIR LANG, 2023).

Para a construção da API GraphQL, utilizou-se a biblioteca Absinthe, cuja documentação oficial detalha desde a criação de schemas até a integração com autenticação e resolvers personalizados (ABSINTHE, 2023). Além disso, práticas recomendadas na construção de APIs foram baseadas nas diretrizes propostas por Fielding (2000), no que tange aos princípios REST, bem como nas especificações da GraphQL Foundation (GRAPHQL FOUNDATION, 2022), especialmente no que se refere ao tratamento de erros e design de schema.

Em termos de arquitetura, adotou-se o padrão Clean Architecture, conforme proposto por Martin (2017), a fim de garantir modularidade, manutenibilidade e separação clara de responsabilidades entre os componentes do sistema. Boas práticas de desenvolvimento, como a organização por contexto, uso de injeção de dependência e testes isolados, também foram aplicadas com base nos princípios do livro “Elixir in Action” (TROYES, 2019) e em tutoriais reconhecidos da comunidade open source.

5. ARQUITETURA DO SISTEMA

Nesta seção, são explorados os aspectos estruturais e arquiteturais que sustentam o backend do WeaveTrip. A seguir, abordaremos os seguintes pontos:

(5.1) o modelo arquitetural adotado, com ênfase na separação de responsabilidades e nas justificativas técnicas que respaldam essa escolha; (5.2) a organização estrutural do projeto, evidenciando como os módulos e pastas foram distribuídos de forma coesa para facilitar o desenvolvimento e a manutenção; (5.3) os diagramas de fluxo e componentes, que representam visualmente tanto a comunicação entre os módulos internos quanto o percurso do usuário em ações-chave do sistema; (5.4) o fluxo de comunicação e integração entre o backend, o frontend, o banco de dados e serviços externos, incluindo os protocolos utilizados e as práticas de segurança; e por fim, (5.5) a análise de impactos e justificativas arquiteturais, destacando como a arquitetura adotada contribui para a escalabilidade, performance e manutenibilidade do sistema.

5.1 Modelo Arquitetural Adotado

O projeto Just Travel Backend adota uma arquitetura modular inspirada em princípios da Clean Architecture, promovendo uma separação clara de responsabilidades entre camadas e módulos. A estrutura da aplicação está dividida em dois domínios principais: weave_trip e weave_trip_web, representando respectivamente a lógica de domínio (core da aplicação) e a camada de interface (entrada/saída).

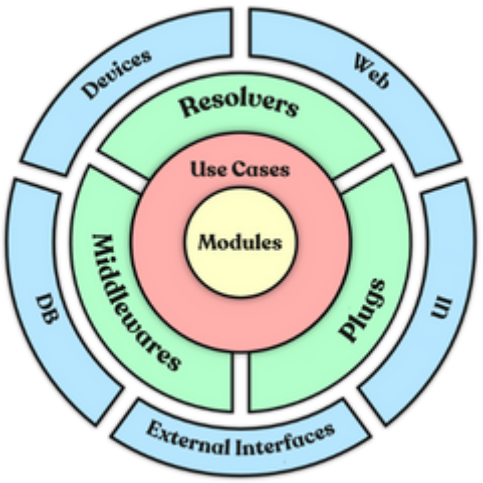

Figura 1: Gráfico Clean Architecture

- **Domínio de negócio (lib/weave_trip)**:

    - Cada funcionalidade principal da aplicação (como media_packages, Reservations, users) é tratada como um módulo independente, com arquivos separados para cada operação CRUD (get.ex, create.ex, update.ex, etc.). Essa abordagem favorece a manutenção, testabilidade e coerência das responsabilidades de cada função.

    - Cada módulo possui um arquivo delegate (ex: media_package.ex, reservation.ex) que centraliza a interface pública do módulo, facilitando o uso pelos resolvers e controllers.

    - Existe um módulo central de acesso a dados (repo.ex), seguindo o padrão do Ecto como gateway para o banco de dados.

- **Interface Web (lib/weave_trip_web)**:

    - Controllers (focados na interface REST, se necessário).

    - Resolvers (interface GraphQL), um para cada módulo (reservation_resolver.ex, travel_packages_resolver. ex, etc.), lidando com a orquestração da lógica de negócio através dos delegates.

    - Middleware com responsabilidades bem definidas como autenticação (authenticate.ex), controle de permissões (check_role.ex) e extração de token (extract_refresh_token.ex).

    - Plugs personalizados que compõem o pipeline de requisição HTTP.

    - O arquivo schema.ex centraliza o schema GraphQL da aplicação, conectando os resolvers e tipos.

A arquitetura adotada neste projeto segue os princípios da Clean Architecture, com uma estrutura modular que promove a separação clara de responsabilidades entre as camadas de domínio, aplicação, infraestrutura e apresentação. Essa escolha técnica oferece uma série de vantagens relevantes no contexto de desenvolvimento backend com Elixir.

Primeiramente, há uma forte ênfase em alta coesão e baixo acoplamento: os módulos de negócio são independentes tanto da interface web quanto do banco de dados, permitindo que a lógica central da aplicação evolua sem dependências externas diretas. Além disso, a estrutura modular — onde cada operação (como get, create, update, delete) está isolada em seu próprio arquivo — facilita significativamente a escrita de testes unitários e de integração, promovendo um desenvolvimento orientado a testes.

Outro ponto importante é a escalabilidade. A modularização por domínio permite a adição de novas funcionalidades ou áreas do sistema de forma organizada, sem impactar negativamente os demais componentes. Isso torna o sistema mais sustentável a longo prazo. A arquitetura também favorece a reutilização da lógica de aplicação por diferentes interfaces — como REST e GraphQL — sem duplicação de código, pois a camada de domínio permanece independente do meio de entrada ou saída. Em resumo, a adoção da Clean Architecture fortalece a manutenção, a clareza e a robustez da aplicação ao longo do seu ciclo de vida.

Essa escolha arquitetural garante uma base sólida, sustentável e extensível para o crescimento contínuo da aplicação Just Travel, permitindo manter a qualidade do código mesmo com o aumento da complexidade do sistema.

5.2 Organização Estrutural do Projeto

📦 JUST-TRAVEL-BACK/

```
├── 📁 config/              # Configurações do projeto
├── 📁 lib/
│   ├── 📁 weave_trip/      # Domínio principal da aplicação
│   │   ├── 📁 auth/            # Autenticação e controle de acesso
│   │   ├── 📁 media_packages/  # Módulo Media Packages
│   │   │   ├── create.ex
│   │   │   ├── get.ex
│   │   │   └── delete.ex
│   │   ├── 📁 payments/        # Módulo de Pagamento
│   │   ├── 📁 Reservations/    # Módulo de Reservas (operações de negócio)
│   │   ├── 📁 travel_packages/ # Módulo Travel Packages
│   │   ├── 📁 users/           # Módulo de usuários
│   │   ├── application.ex      # Módulo principal de integração da aplicação
```

```
│   │   ├── repo.ex              # Repositório (interface com o banco)
│   │   ├── upload.ex            # Lógica de upload
│   │   └── users.ex             # Entrypoint de domínio do usuário
│   │
│   ├── 📁 weave_trip_web/     # Camada de apresentação (Web e GraphQL)
│   │   ├── 📁 controllers/      # (em branco ou reservado)
│   │   ├── 📁 middleware/       # Middlewares de autenticação, roles, etc.
│   │   ├── 📁 plugs/            # Plugs customizados para o pipeline do Phoenix
│   │   ├── 📁 resolvers/        # Resolvers GraphQL por domínio
│   │   ├── context.ex           # Contexto geral da aplicação
│   │   ├── endpoint.ex          # Endpoint Phoenix
│   │   ├── router.ex            # Rotas da aplicação
│   │   ├── schema.ex            # Esquema GraphQL principal
│   │   └── weave_trip_web.ex    # Entrypoint web
│
├── 📁 priv/
│   ├── 📁 repo/
│   │   ├── 📁 migrations/       # Scripts de migração do banco de dados
│   │   └── seeds.exs            # Scripts de seed e dados iniciais
│
├── 📁 test/                 # Diretório de testes automatizados
├── 📁 docs/                 # Documentação técnica do sistema
└── 📁 static/               # Arquivos estáticos, se aplicável
```

5.3 Funcionalidades, Módulos e Regras de negócio

Nesta seção, são apresentadas as principais funcionalidades, módulos implementados e respectivas regras de negócio da API. Cada subseção detalha uma entidade central do sistema, descrevendo suas responsabilidades, comportamentos esperados, fluxos de operação e restrições aplicáveis. Os tópicos abordados incluem: (1) Users, com foco na gestão de usuários e controle de autenticação e autorização; (2) Travel Package, referente à criação, listagem e gerenciamento de pacotes de viagem; (3) Media Package, que trata do armazenamento e associação de mídias aos pacotes; (4) Reservation, que abrange o processo de reserva de pacotes, controle de vagas e políticas de cancelamento; e (5) Payment, que centraliza o processo de pagamentos relacionados às reservas, abrangendo desde a criação e agrupamento de intenções de pagamento com a Stripe até a confirmação, cancelamento e gestão de métodos de pagamento,

garantindo segurança, rastreabilidade e integração eficiente com a infraestrutura financeira da aplicação.

5.3.1 Users

A entidade User representa os usuários do sistema, que podem assumir diferentes papéis (como client, agency ou admin) e interagir com as demais funcionalidades da plataforma. A modelagem foi desenvolvida com foco em segurança, consistência de dados, regras de permissão e usabilidade, alinhando-se às boas práticas de autenticação e controle de acesso.

Atributos Principais

Campo	Tipo	Descrição
id	**id**	Identificador único do usuário.
first_name	**string**	Primeiro nome.
last_name	**string**	Sobrenome (opcional).
email	**string**	Endereço de e-mail único e válido.
password	**string**	Senha do usuário (virtual, usada apenas para criação/alteração).
password_hash	**string**	Hash seguro da senha, gerado com Pbkdf2.
document	**string**	CPF ou CNPJ válidos, verificados com Brcpfcnpj.
role	**string**	Papel do usuário no sistema: client, agency, admin, etc.
birthdate	**string**	Data de nascimento.
phone	**string**	Número de telefone com DDD.
cep, street, neighborhood, house_number, city, state	**string**	Campos de endereço completo.
inserted_at	**timestamp**	Data de criação do registro.

updated_at	**timestamp**	Data da última modificação.

Regras de Negócio

A seguir estão listadas as principais regras de negócio associadas à manipulação de usuários:

- **Validação de E-mail**: Deve ser único e seguir o formato padrão (e.g. user@example.com).
- **Documento**: Obrigatório e deve ser um CPF ou CNPJ válido. Validado com a biblioteca Brcpfcnpj.
- **Senha**: Não é armazenada em texto plano; seu hash é gerado com Pbkdf2.
- **Roles e Permissões**:
    - Apenas usuários com role: "admin" podem alterar o papel de outros usuários para "admin".
    - Um usuário só pode alterar seus próprios dados, a menos que seja um administrador.
- **Atributos Obrigatórios**: first_name, email, password, document e role (com validações específicas).
- **Autenticação**: Utiliza JWT com suporte a Access Token e Refresh Token.
- **Autorização**: Implementada em cada resolver de acordo com a função do usuário logado.

Módulos e Funcionalidades

A estrutura do módulo de usuários segue o padrão de separação por contexto e responsabilidade, com os seguintes componentes principais:

1. **Users.List**

Permite listar usuários com suporte a paginação, ordenação e filtros. Os parâmetros disponíveis incluem limit para definir a quantidade de registros retornados, offset para indicar o ponto inicial da listagem, order_by para escolher o campo de ordenação (como first_name, email ou inserted_at), e direction para definir a direção da ordenação (asc ou desc). Caso não sejam informados os parâmetros de ordenação, a listagem será feita por first_name em ordem crescente.

- Parâmetros: limit, offset, order_by, direction.
- Ordenação padrão: first_name ASC.

2. **Users.Get**

Busca um usuário por ID fornecido. Caso o ID não seja numérico, retorna o erro "Invalid ID format". Se o usuário não for encontrado no banco, retorna o erro "User not found". Em caso de sucesso, retorna os dados completos do usuário correspondente ao ID informado.

3. **Users.Create**

Realiza a criação de um novo usuário aplicando validações rigorosas de integridade e segurança. Verifica se os campos obrigatórios first_name, last_name, email, password, document e role estão presentes. O e-mail é validado quanto ao formato e unicidade, o documento é validado como CPF ou CNPJ válido, e o campo role aceita apenas os valores "client", "agency" ou "admin". A senha é criptografada com Pbkdf2 antes de ser armazenada. Em caso de erro, são retornadas mensagens claras indicando falhas específicas como e-mail duplicado ou documento inválido.

4. **Users.Update**

Permite atualização parcial ou total dos dados do usuário, respeitando permissões e validações. Usuários comuns podem atualizar apenas seus próprios dados e não podem alterar seu papel para "admin". Administradores podem atualizar qualquer usuário, inclusive a si mesmos.

O sistema verifica se o usuário existe, se o token JWT está ativo, e se o usuário atual tem permissão para a ação, retornando erro "Unauthorized to update this user" quando aplicável. Validações incluem verificação de e-mail duplicado, CPF/CNPJ válido, campos obrigatórios, ID válido e regras específicas para o campo role. Em caso de sucesso, retorna os dados atualizados do usuário.

5. **Users.Delete**

Remove o registro de um usuário pelo ID, verificando previamente sua existência. Se o usuário não for encontrado, retorna erro "Failed to delete user". Em caso de exclusão bem-sucedida, retorna a mensagem "User deleted successfully".

6. **Users.ChangePassword**

Realiza a troca segura de senha do usuário, validando a senha atual com Pbkdf2. Exige confirmação da nova senha para garantir que a senha nova e sua confirmação sejam iguais. Aplica políticas mínimas de força, como tamanho mínimo de 6 caracteres. A nova senha é criptografada antes de ser salva. Retorna erro em caso de senha atual inválida, confirmação diferente ou falha na validação. Em caso de sucesso, retorna os dados do usuário atualizado.

7. **Users.Login**

Permite que o usuário realize login com email e senha válidos. Utiliza o resolver AuthResolver.login/3, que chama o método authenticate/2 do módulo Guardian para verificar se o usuário existe e validar a senha com Pbkdf2.verify_pass. Em caso de sucesso, gera um Access Token com validade de 15 minutos e um Refresh Token com validade de 7 dias. Retorna um objeto contendo ambos os tokens. Em caso de falha na autenticação (usuário inexistente ou senha incorreta), retorna mensagem clara de erro.

8. **Users.Refresh_token**

Gera um novo Access Token válido a partir de um Refresh Token ativo. Utiliza o resolver AuthResolver.refresh_token/3, que decodifica e verifica o refresh token com Guardian.decode_and_verify/1. Se o token for válido e o usuário identificado, gera um novo Access Token com validade de 15 minutos. Implementa rotação de tokens, garantindo segurança. Se o refresh token for inválido, expirado ou a verificação falhar, retorna erro com a mensagem "Invalid refresh token".

9. **Users.Logout**

Realiza a revogação segura do Access Token ativo utilizando o resolver AuthResolver.logout/3. O token de acesso presente no contexto da requisição é revogado pelo método Guardian.revoke/1, tornando-o inválido para futuras requisições. Em caso de sucesso, retorna mensagem de confirmação de logout. Se ocorrer erro na revogação, como token ausente ou já inválido, retorna mensagem de erro adequada.

5.3.2 Travel Package

A entidade TravelPackage representa os pacotes de viagem disponibilizados na plataforma WeaveTrip. Cada pacote é associado a um usuário, que pode ser um administrador ou uma agência, responsáveis pela criação e gerenciamento dos pacotes. A modelagem dessa entidade prioriza a integridade dos dados, validações robustas e suporte às operações essenciais como criação, atualização, consulta, exclusão e gerenciamento de mídia associada.

O schema inclui campos como nome, destino, preço, datas de início e fim, disponibilidade, descrição, número de vagas restantes, status, política de cancelamento e um campo virtual para a thumbnail do pacote. O relacionamento com usuários é feito via belongs_to, enquanto a associação com mídias (fotos e vídeos) é mantida via has_many, com os links das mídias armazenados externamente (por exemplo, no MinIO). A persistência dos arquivos multimídia ocorre fora do banco, mantendo apenas as URLs no banco de dados.

Atributos Principais

Campo	Tipo	Descrição
id	**integer**	Identificador único do pacote de viagem (gerado automaticamente).
users_id	**integer**	Referência ao usuário (admin ou agência) responsável pelo pacote.
destination	**string**	Destino do pacote de viagem.
name	**string**	Nome do pacote de viagem.
price	**decimal**	Preço do pacote de viagem.
description	**text**	Descrição detalhada do pacote de viagem.
start_date	**utc_datetime**	Data e hora de início da viagem.
end_date	**utc_datetime**	Data e hora de término da viagem.
availability	**integer**	Número total de vagas disponíveis no pacote.
cancellation_policy	**string (opcional)**	Política de cancelamento: "flexible", "moderate", "strict" ou vazio.

remaining_slots	**integer**	Número de vagas restantes no pacote de viagem.
status	**string**	Status atual do pacote: "active", "inactive" ou "cancelled".
thumbnail	**string (virtual)**	URL da imagem ou miniatura representativa do pacote. Esse campo é virtual e não é persistido no banco de dados.
inserted_at	**utc_datetime**	Data e hora de criação do registro.
updated_at	**utc_datetime**	Data e hora da última atualização do registro.

Regras de Negócio

- **Disponibilidade de Vagas:** O sistema garante que o número total de reservas feitas para um pacote de viagem não ultrapasse a quantidade de vagas disponíveis, representada pelo campo remaining_slots (ou available_slots). Ao realizar uma nova reserva, valida que ainda há vagas suficientes. Caso contrário, bloqueia a reserva para evitar overbooking.
- **Validação do Preço:** O valor do preço (price) do pacote é sempre positivo e representado com precisão de até duas casas decimais. Isso assegura que o preço é válido e evita erros ou valores inconsistentes na cobrança.
- **Validação das Datas:** A data final (end_date) do pacote é obrigatoriamente posterior à data de início (start_date). Essa regra evita pacotes com duração inválida ou que terminem antes de começarem, garantindo a coerência do período da viagem.
- **Título do Pacote:** O campo name (título) é obrigatório para identificação do pacote e é único entre os pacotes com status ativo (status == "active"). Isso evita confusões na plataforma e permite que os usuários encontrem e identifiquem os pacotes facilmente.
- **Restrições para Alterações e Cancelamentos:** Pacotes só podem ser atualizados ou cancelados quando não existirem reservas confirmadas vinculadas a eles. Caso haja reservas ativas, alterações que impactem a

viagem são bloqueadas para proteger os direitos dos clientes e garantir a estabilidade dos contratos firmados.

- **Mídia Associada:** Múltiplos arquivos de mídia podem ser adicionados ao criar ou atualizar o pacote, sendo que o envio dessas mídias deve ser feito via API REST. Uma mídia pode ser marcada como primária (is_primary). O thumbnail do pacote é a URL da mídia primária ou um valor padrão (fallback), caso não exista nenhuma mídia primária associada. O upload das mídias deve ser realizado em um bucket específico, chamado "travel-packages". Caso ocorra falha no upload ou na criação da mídia, toda a transação é revertida para garantir a integridade dos dados.

Módulos e Funcionalidades

A estrutura do módulo de Pacotes de Viagem segue o padrão de separação por contexto e responsabilidade, com os seguintes componentes principais:

1. **TravelPackages.List**

O módulo TravelPackages.List permite listar pacotes de viagem com suporte a filtros, paginação e ordenação. Os parâmetros disponíveis para a listagem são: limit, que define o número máximo de registros retornados; offset, que indica o ponto inicial da listagem para possibilitar paginação; order_by, que especifica o campo para ordenar os resultados (como price, start_date, destination, entre outros campos válidos); e direction, que define a direção da ordenação, podendo ser ascendente (asc) ou descendente (desc).

A ordenação é aplicada somente se o campo especificado em order_by estiver entre os campos válidos predefinidos no sistema, caso contrário, utiliza-se um campo padrão (name). Além disso, ao buscar os pacotes, o sistema realiza uma junção com as mídias associadas para identificar a mídia marcada como primária e retorna sua URL para ser usada como thumbnail do pacote. Caso não exista uma mídia primária, é fornecida uma imagem placeholder padrão.

Os pacotes retornados incluem a URL do thumbnail, e os valores monetários, como o preço (price), são convertidos para o formato float para facilitar o consumo pela camada de apresentação. O módulo também disponibiliza uma função para contar o total de pacotes disponíveis, útil para controle e paginação na interface.

2. **TravelPackages.Get**

O módulo TravelPackages.Get é responsável por buscar um pacote de viagem específico com base no seu ID. Quando uma requisição é feita, o sistema tenta localizar o pacote no banco de dados e, ao mesmo tempo, carrega (faz preload) os dados relacionados, como o usuário responsável pelo pacote e as mídias associadas. Caso o ID fornecido não corresponda a nenhum registro existente, o sistema retorna um erro indicando que o pacote não foi encontrado, com uma mensagem clara para o usuário: "Pacote não encontrado" e detalhes informando que o ID informado não corresponde a nenhum pacote existente. Em caso de sucesso, todos os dados do pacote são retornados, incluindo as mídias associadas, e o valor do preço é convertido de decimal para número de ponto flutuante para facilitar o consumo da informação. Não há uma validação explícita do formato do ID no código apresentado, mas a ausência do registro é tratada de forma clara.

3. **TravelPackages.Create**

O módulo TravelPackages.Create realiza a criação de um novo pacote de viagem, garantindo que todas as validações de integridade e regras de negócio sejam respeitadas. Os campos obrigatórios incluem título (title), descrição, preço (price), duração em dias (duration_days), destino (destination), data de início (start_date), data de fim (end_date), número de vagas disponíveis (available_slots) e itens inclusos (included_items). Antes da inserção, o sistema valida que o título não está duplicado, que o preço é positivo e que a data final é posterior à data inicial, entre outras regras definidas no changeSet. Em caso de falhas, mensagens específicas são retornadas, como "Title already exists" para título duplicado ou "Invalid date range" para datas inválidas.

Além disso, o processo de criação é realizado dentro de uma transação para garantir a integridade dos dados. Caso o pacote seja criado com sucesso, o sistema também realiza o upload de múltiplos arquivos de mídia, armazenando-os em um bucket específico chamado "travel-packages". Cada mídia pode ser marcada como primária para servir como thumbnail do pacote. Se qualquer falha ocorrer durante o upload ou a criação das mídias, toda a transação é revertida, mantendo o banco consistente. Após a criação, o pacote é recarregado com suas mídias associadas e o thumbnail é definido com base na mídia primária ou em uma imagem padrão de

fallback. No resolver GraphQL, o preço é convertido para número decimal flutuante, e erros de validação são traduzidos em mensagens claras para o usuário.

4. **TravelPackages.Update**

O módulo WeaveTrip.TravelPackages.Update é responsável por realizar a atualização parcial ou total de um pacote de viagem, com diversas validações e restrições importantes. A atualização pode ser realizada por dois caminhos distintos: pela agência responsável (call/3), que exige a verificação da autoria do pacote (checando se o users_id do pacote corresponde ao user_id do usuário autenticado), ou pelo administrador (call_admin/2), que possui permissão irrestrita e pode atualizar qualquer pacote.

Um pacote só poderá ser atualizado caso o usuário tenha as devidas permissões, caso contrário será retornado um erro com :unauthorized ou :not_found quando o pacote não for encontrado. No fluxo de atualização, são verificadas também as regras de integridade dos dados, como a unicidade do título (não mostrada diretamente no código enviado, mas que pode ser implementada no changeset_update/2), além da consistência entre as datas (start_date e end_date), e a política de cancelamento, todas validadas no changeset.

Após validar e atualizar os campos principais do pacote, o sistema também permite adicionar ou modificar as mídias associadas, através da função update_media_package/2, que gerencia o upload dos arquivos, inserção dos registros no banco de dados e rollback da transação caso ocorra alguma falha.

Além disso, o código implementa um sistema robusto de validações através do TravelPackage.changeset_update/2, que garante que os dados estão completos e corretos, como a obrigatoriedade de campos essenciais, valores permitidos para status e cancellation_policy, e o formato adequado do name. A validação validate_user_permission/2 também reforça a segurança, garantindo que apenas admins ou agencies possam criar ou editar pacotes.

No TravelPackageResolver, o endpoint GraphQL que expõe a funcionalidade trata adequadamente as permissões, distinguindo o fluxo de atualização entre admin e agency. O código também converte o preço do pacote (Decimal para float) para manter a compatibilidade com o formato esperado pela API.

5. **TravelPackages.Delete**

O módulo TravelPackages.Delete é responsável pela remoção de pacotes de viagem do sistema, identificados pelo seu ID. Antes da exclusão efetiva, o sistema executa uma rotina que remove todas as mídias associadas ao pacote, garantindo a integridade e evitando arquivos órfãos no armazenamento em nuvem. Para usuários do tipo "agency", o sistema valida a autoria do pacote: apenas a agência que criou o pacote pode removê-lo. Já administradores ("admin") podem excluir qualquer pacote, sem necessidade de validação de autoria.

No fluxo de exclusão, caso o pacote não exista, o sistema retorna o erro {:error, :not_found} com a mensagem apropriada. Se o usuário não for autorizado a excluir o pacote, é retornado {:error, :unauthorized}.

5.3.3 Media Package

O Media Package representa os arquivos de mídia (como imagens e vídeos) associados a um determinado Pacote de Viagem (Travel Package). Esse módulo é responsável por armazenar e gerenciar as mídias que ilustram, promovem ou descrevem visualmente um pacote de viagem.

Cada mídia possui atributos como legenda, URL do arquivo e uma indicação de se é a imagem principal do pacote. As mídias são vinculadas diretamente ao pacote de viagem ao qual pertencem, respeitando a integridade referencial e regras de autorização para operações sensíveis.

Atributos Principais

Atributo	Tipo	Descrição
id	**Inteiro**	Identificador único da mídia
travel_package_id	**Inteiro**	ID do pacote de viagem ao qual a mídia está associada
file_caption	**String**	Legenda descritiva da mídia
url_file	**Texto**	URL pública do arquivo de mídia armazenado
is_primary	**Booleano**	Indica se a mídia é a imagem principal do pacote

inserted_at	**DateTime**	Timestamp de criação da mídia
updated_at	**DateTime**	Timestamp de última atualização da mídia

Regras de Negócio

- Associação Obrigatória ao Pacote de Viagem: Toda mídia deve obrigatoriamente estar associada a um travel_package_id válido. Se o pacote for excluído, todas as mídias associadas também são removidas automaticamente (regra on_delete: :delete_all).
- **Controle de mídia principal:** Cada Pacote de Viagem pode conter diversas mídias, porém apenas uma deve ser marcada como principal (is_primary: true). O sistema não impede múltiplas mídias marcadas como principais, mas é recomendado implementar validação na camada de negócio para garantir que apenas uma seja destacada como principal.
- **Armazenamento e acesso à mídia:** Os arquivos de mídia são armazenados externamente, por exemplo, em buckets S3. A URL pública de acesso (url_file) é persistida no banco de dados. O sistema manipula a URL para extrair o object_key ao realizar operações como exclusão física do arquivo no bucket.
- **Exclusão automática na remoção do Pacote de Viagem:** Existe uma operação alternativa exclusiva para administradores, que permite a exclusão sem verificação de propriedade.
- **Criação em lote:** É possível criar múltiplas mídias associadas a um único Pacote de Viagem por meio do método changeset_create_multiple. Cada mídia deve incluir obrigatoriamente url_file, file_caption e is_primary.

Módulos e Funcionalidades

O módulo Media Package é responsável pela gestão das mídias associadas aos pacotes de viagem, permitindo o cadastro, consulta, atualização e exclusão de arquivos de mídia (imagens). Esse módulo garante que cada pacote de viagem possua imagens ilustrativas, organizadas e gerenciadas com segurança e eficiência.

1. **Criação de Mídias Associadas a Pacotes de Viagem**

A criação de mídias associadas a pacotes de viagem permite que uma ou múltiplas mídias sejam vinculadas a um pacote específico. Cada mídia cadastrada deve conter obrigatoriamente informações como a legenda (file_caption), a URL do arquivo armazenado (url_file) e a indicação se ela representa a imagem principal do pacote (is_primary). Esse processo é realizado através da função changeset_create, responsável por validar os dados fornecidos e inserir o registro de forma segura no banco de dados.

Além disso, foi implementado o changeset_create_multiple, que possibilita a inserção em lote de várias mídias associadas a um mesmo pacote de viagem, facilitando e agilizando o processo de cadastro quando há múltiplas imagens para um mesmo pacote. Todo o processo garante a integridade referencial através do atributo travel_package_id, assegurando que cada mídia esteja sempre corretamente vinculada a um pacote de viagem existente.

2. **Consulta de Mídias (GET)**

A funcionalidade de consulta de mídias permite recuperar todas as informações detalhadas de uma mídia específica a partir do seu identificador único (id). Esse processo é realizado por meio da função Get.call(id), que efetua a busca na base de dados e retorna os dados completos caso a mídia seja encontrada. As informações retornadas incluem o ID da mídia, as datas de inserção e de atualização do registro, a legenda associada (file_caption), a URL do arquivo armazenado (url_file), a indicação se a mídia é a principal do pacote (is_primary) e, por fim, o ID do pacote de viagem ao qual a mídia está vinculada.

3. **Exclusão de Mídias (DELETE)**

A funcionalidade de exclusão de mídias permite remover uma mídia do sistema de forma segura e controlada. O processo envolve, inicialmente, a validação de autorização, garantindo que apenas o proprietário do pacote de viagem associado ou administradores do sistema possam realizar a exclusão. Em seguida, ocorre a remoção física do arquivo armazenado no bucket S3, assegurando que o recurso não permaneça disponível após a exclusão.

Por fim, realiza-se a exclusão lógica do registro correspondente na base de dados, eliminando a referência à mídia. Para executar essa operação, existem duas funções distintas: Delete.call(media_package_id, user_id), que realiza a exclusão mediante a validação de propriedade do pacote, e Delete.call_admin (media_package_id), que permite a exclusão direta, sendo utilizada exclusivamente por administradores.

4. **Atualização de Mídias**

A atualização de informações de uma mídia existente permite modificar atributos como a legenda (file_caption), a URL do arquivo (url_file), ou ainda definir qual será a mídia principal (is_primary) associada a um pacote de viagem. Este processo é realizado por meio da função changeset_update, que garante a validação de todas as alterações conforme as regras de negócio e os tipos definidos no schema MediaPackage.

O modelo também permite indicar qual mídia é a principal (is_primary: true), como demonstrado no seed TravelPackagesSeed, especialmente ao criar múltiplas imagens para um pacote, como no caso do "Roma Imperial". Esse aspecto evidencia que, ao atualizar uma mídia, é importante garantir que apenas uma seja marcada como principal por pacote, o que pode demandar lógica adicional de validação na atualização.

5. **Upload e Armazenamento de Arquivos**

O sistema realiza o upload e o armazenamento das imagens associadas aos pacotes de viagem em um bucket S3, utilizando a pasta lógica "travel-packages" para organização dos arquivos. Para o envio e remoção de arquivos, é utilizado o módulo WeaveTrip.Upload, que centraliza as operações de interação com o serviço S3, garantindo consistência e segurança nos processos.

Durante o upload, os arquivos podem ser enviados com ou sem adição de prefixos como UUIDs, a depender da necessidade. Por exemplo, conforme implementado no módulo PlaceholderThumbnailSeed, é possível realizar uploads mantendo o nome original do arquivo — como no caso do envio do thumbnail.jpg diretamente para o bucket placeholder. O procedimento inclui verificações importantes, como a confirmação da existência do arquivo localmente antes do

envio, e a tentativa automática de criar o bucket, caso este ainda não exista. Após o upload, o sistema gera e disponibiliza a URL pública do arquivo, facilitando seu uso posterior.

No contexto da exclusão de arquivos, a URL armazenada no banco de dados é processada para extrair a object_key correspondente no S3, o que permite realizar a remoção precisa e segura do objeto no bucket, evitando falhas ou exclusões incorretas.

O armazenamento das informações referentes às mídias no banco de dados é gerenciado pela tabela media_packages, representada pelo módulo WeaveTrip.MediaPackages.MediaPackage. Essa tabela contém os campos url_file (armazenando a URL pública do arquivo), file_caption (descrição da mídia) e is_primary (indicando se a mídia é a principal do pacote). A associação com o pacote de viagem (travel_package) é realizada através de uma chave estrangeira.

O módulo também implementa funções robustas para criação (changeset_create) e atualização (changeset_update) dos registros, garantindo validação obrigatória dos dados essenciais. Além disso, a função changeset_create_multiple permite a criação eficiente de múltiplos registros de mídia relacionados a um mesmo pacote de viagem, utilizando listas de entrada que são mapeadas em uma sequência de changesets válidos.

5.3.4 Reservation

A entidade Reservation representa a efetivação de uma reserva realizada por um cliente para um pacote de viagem, mediada por uma agência. Esta entidade é essencial para o controle de disponibilidade dos pacotes e para o gerenciamento do fluxo de confirmação, cancelamento ou expiração das reservas. Além disso, é responsável por armazenar dados críticos como o número de viajantes, valor total e a política de cancelamento aplicável.

Atributos Principais

Campo	Tipo	Descrição
id	**id**	Identificador único da reserva.
client_id	**id**	Referência ao cliente que realizou a reserva.

package_id	**id**	Referência ao pacote de viagem reservado.
agency_id	**id**	Referência à agência responsável pelo pacote.
status	**string**	Estado atual da reserva: "PENDING", "CONFIRMED", "CANCELLED", "CANCELED", "EXPIRED" ou "REFUNDED".
token	**uuid**	Identificador único da reserva, gerado automaticamente para rastreamento seguro.
reservation_date	**date**	Data em que a reserva foi realizada.
expiration_date	**date**	Data limite para confirmação ou pagamento da reserva.
payment_method	**string**	(Opcional) Forma de pagamento escolhida pelo cliente.
total_price	**decimal**	Valor total da reserva. Não pode ser negativo.
traveler_count	**integer**	Número de viajantes incluídos na reserva. Deve ser maior que zero.
cancellation_policy	**string**	Texto descritivo sobre a política de cancelamento aplicada à reserva.

Regras de Negócio

- **Campos obrigatórios na criação:** Os seguintes campos são obrigatórios ao criar uma reserva: client_id, package_id, agency_id, status, token, reservation_date e expiration_date. Isso assegura que cada reserva possua todas as informações essenciais para ser validamente processada.
- **Token único é gerado automaticamente:** O campo token é um UUID utilizado para identificar de forma única cada reserva, garantindo rastreabilidade e segurança em operações como confirmação ou cancelamento. Caso não seja fornecido, o sistema gera automaticamente.
- **Status limitado a valores permitidos:** O campo status só pode conter um dos seguintes valores:
    - "PENDING": Reserva aguardando confirmação ou pagamento.

    - "CONFIRMED": Reserva confirmada e efetivada.
    - "CANCELLED" ou "CANCELED": Reserva cancelada pelo cliente, agência ou sistema. Ambas variações são aceitas no processo de atualização para garantir flexibilidade.
    - "EXPIRED": Reserva que ultrapassou o prazo sem confirmação.
    - "REFUNDED": Reserva que foi cancelada com devolução dos valores pagos.

- **Contagem de viajantes obrigatoriamente positiva:** O campo traveler_count deve conter um número maior que zero, refletindo o número real de pessoas que utilizarão o pacote reservado. Essa regra evita reservas inconsistentes ou inválidas.
- **Valor total não pode ser negativo:** O campo total_price representa o valor final da reserva, devendo ser sempre igual ou superior a zero. O sistema rejeita reservas com valores negativos, garantindo integridade financeira.
- **Restrições de integridade referencial:** As chaves estrangeiras client_id, package_id e agency_id são validadas pelo sistema, assegurando que o cliente, o pacote e a agência referenciados existam no banco de dados. Isso impede a criação de reservas órfãs ou inválidas.
- **Prevenção de duplicação de reservas pendentes:** Quando o cliente tenta realizar uma nova reserva para um mesmo pacote, e já existe uma reserva com status "PENDING" associada ao mesmo cliente e pacote, o sistema não cria uma nova reserva. Em vez disso, atualiza a reserva existente, ajustando os campos traveler_count e total_price conforme os novos dados fornecidos. Esta regra evita acúmulo de reservas duplicadas e garante um fluxo mais limpo de reservas pendentes.
- **Redução automática das vagas ao confirmar a reserva:** Quando uma reserva tem seu status alterado para "CONFIRMED", o sistema automaticamente reduz a quantidade de vagas disponíveis no pacote de viagem correspondente, considerando o valor de traveler_count. Antes de

realizar essa alteração, o sistema valida se existem vagas suficientes disponíveis. Caso não haja, a confirmação é impedida e um erro é retornado.

- **Restauração de vagas ao desfazer a confirmação:** Se uma reserva confirmada for posteriormente alterada para qualquer outro status, como "CANCELLED", "EXPIRED" ou "REFUNDED", o sistema restaura automaticamente as vagas ao pacote de viagem. Isso evita bloqueio indevido de vagas e mantém a disponibilidade atualizada.

- **Atualização de campos com restrições:** Na atualização de uma reserva, a maioria dos campos pode ser modificada, incluindo status, traveler_count e payment_method. No entanto, o campo id é imutável, preservando a integridade da identificação da reserva ao longo do tempo.

- **Validação de disponibilidade ao confirmar reserva:** Antes de permitir a transição de uma reserva para o status "CONFIRMED", o sistema verifica se o pacote possui vagas suficientes para acomodar o número de viajantes especificado em traveler_count. Se a disponibilidade for insuficiente, a confirmação é rejeitada para evitar sobrecarga de reservas.

- **Controle de acesso e segurança:** Somente o cliente responsável pela reserva, a agência proprietária do pacote ou um administrador do sistema possuem permissão para visualizar ou modificar os dados da reserva. Isso garante confidencialidade e segurança no tratamento das informações.

Módulos e Funcionalidades

A estrutura do módulo Reservation foi concebida para representar o conceito de um carrinho virtual especializado na reserva de pacotes de viagem, funcionando como o elo entre cliente, agência e pacote. Diferente de um carrinho genérico, onde itens podem ser adicionados ou removidos livremente, aqui a reserva possui regras de disponibilidade, confirmação e pagamento, que garantem a integridade do processo de compra. O módulo é segmentado em componentes com responsabilidades bem definidas, assegurando clareza, manutenção e evolução contínua da aplicação.

1. **Reservation.List**

O módulo Reservation.List é responsável por implementar a funcionalidade de listagem de reservas com suporte a paginação e ordenação dinâmica. Ele expõe uma interface para recuperar múltiplas reservas a partir de critérios configuráveis, garantindo que os dados sejam apresentados de forma eficiente e consistente.

Quando uma requisição de listagem é realizada via GraphQL, o resolver ReservationResolver.list_reservations/3 delega a operação ao módulo Reservations.List. Este, por sua vez, utiliza as funcionalidades do Ecto.Query para aplicar os filtros recebidos:

- **limit**: Define a quantidade máxima de reservas retornadas, com valor padrão de 10.
- **offset**: Determina a partir de qual posição os registros serão recuperados, permitindo a implementação de paginação (padrão: 0).
- **order_by**: Estabelece o campo pelo qual a listagem será ordenada; caso omitido ou inválido, o campo padrão é o id.
- **direction**: Define a direção da ordenação, que pode ser "asc" (ascendente) ou "desc" (descendente), com padrão em "asc".

2. **Reservation.Get**

O módulo Reservation.Get tem como responsabilidade principal a recuperação de uma reserva específica com base em seu identificador único (ID), garantindo que as regras de acesso e segurança sejam rigorosamente aplicadas.

O fluxo começa com a chamada ao resolver ReservationResolver.get_reservation/3, que encaminha a requisição ao módulo Reservations.Get. Este módulo executa a busca pela reserva no banco de dados e valida o contexto do usuário autenticado para assegurar que apenas perfis autorizados possam visualizar as informações.

A lógica de acesso implementada é a seguinte:

- Se o usuário for do tipo client, é verificado se ele é proprietário da reserva. Caso não seja, a operação é negada com erro "Unauthorized".

- Para usuários do tipo admin ou agency, a reserva pode ser consultada livremente, independentemente do cliente associado.
- Se a reserva não for encontrada no banco de dados, é retornado erro 404: "Reservation not found".
- Se não houver autenticação válida ou o perfil não for permitido, o sistema retorna erro: "Unauthorized".

3. **Reservations.Create**

O módulo Reservations.Create é responsável por orquestrar todo o processo de criação de uma reserva, garantindo a consistência de dados e o controle adequado de disponibilidade de vagas. Esse processo pode resultar na criação de uma nova reserva ou na atualização de uma reserva pendente já existente, evitando duplicações desnecessárias. O fluxo inicia com o recebimento dos parâmetros essenciais, como o ID do cliente, do pacote de viagem, da agência, o número de viajantes e o status inicial (tipicamente "PENDING"). Após o recebimento, realiza-se a validação e a recuperação do pacote, assegurando que ele exista e que o ID seja válido.

Em seguida, há uma verificação rigorosa da disponibilidade de vagas, onde o sistema compara as vagas restantes no pacote com a quantidade de viajantes desejada. Se não houver vagas suficientes, a operação é imediatamente rejeitada. Caso contrário, o sistema avalia se já existe uma reserva pendente com as mesmas características (cliente, pacote, agência). Se não existir, uma nova reserva é criada, calculando-se o valor total com base no preço do pacote e na quantidade de viajantes. Se já houver uma reserva pendente, esta é atualizada com as novas informações, atuando efetivamente como um “carrinho de compras”.

Se a reserva for confirmada ("CONFIRMED") e não se tratar de uma atualização, o sistema realiza um ajuste no pacote, reduzindo as vagas disponíveis conforme o número de viajantes. Além disso, existe um fluxo especializado para a criação de reservas temporárias, onde o sistema gera um token único e estabelece uma data de expiração automática de 24 horas. Essas reservas temporárias são úteis para garantir a disponibilidade antes da confirmação do pagamento, mas não impactam imediatamente no número de vagas disponíveis, prevenindo casos de overbooking.

Durante todo o processo, são aplicadas diversas validações, como a obrigatoriedade de campos, consistência relacional entre cliente, agência e pacote, e restrições numéricas. Esse módulo retorna sempre um resultado explícito, indicando o sucesso ou erro da operação, com mensagens claras e informativas para o usuário ou sistemas integrados.

4. **Reservations.Update**

O módulo Reservations.Update gerencia as operações de modificação de reservas existentes, garantindo não apenas a integridade dos dados, mas também a correta gestão das vagas disponíveis nos pacotes associados. A atualização de uma reserva deve respeitar um conjunto de validações rigorosas: o id da reserva não pode ser alterado, o package_id deve necessariamente existir, e o status informado precisa estar dentro dos valores permitidos — "PENDING", "CONFIRMED", "CANCELED", "EXPIRED" ou "REFUNDED".

A atualização do status de uma reserva está atrelada a regras específicas de negócio. Por exemplo, ao alterar o status para "CONFIRMED", o sistema verifica se há vagas suficientes no pacote para acomodar a quantidade de viajantes solicitada. Se uma reserva previamente confirmada for modificada para outro status, o sistema devolve as vagas correspondentes ao pacote, garantindo um controle preciso do inventário. Quando a reserva permanece como "CONFIRMED" sem alterações na quantidade de viajantes, o sistema mantém o número de vagas inalterado.

Toda a operação é realizada de forma transacional, utilizando Repo.transaction/1, o que garante que, em caso de falha em qualquer etapa, nenhuma modificação será persistida no banco de dados. Possíveis erros são tratados explicitamente, como :not_found para reservas inexistentes, :package_not_found para pacotes inválidos, além de validações de integridade como proibição de alterar o id e a verificação da disponibilidade de vagas. Este módulo também é preparado para acionar processos subsequentes, como o processamento de pagamento, após a confirmação de uma reserva, tornando-se um elo fundamental entre a validação e a finalização da transação comercial.

5. **Reservations.Delete**

O processo de exclusão de uma reserva no sistema WeaveTrip é realizado através do resolver ReservationResolver.delete_reservation/3, que invoca o módulo Reservations.Delete para executar a operação. Inicialmente, o sistema tenta localizar a reserva com base no ID fornecido. Caso a reserva não seja encontrada, o sistema responde com uma mensagem de erro clara: "Reserva não encontrada". Se a reserva for localizada e a exclusão for realizada com sucesso, o sistema retorna a mensagem: "Reserva excluída com sucesso".

No entanto, se ocorrer algum erro inesperado durante o processo, o sistema responde com uma mensagem genérica indicando a falha: "Erro ao excluir reserva". A exclusão é uma ação definitiva e, por isso, o sistema assegura que todas as validações necessárias sejam realizadas antes da remoção do registro.

5.3.5 Payment

A entidade Payment representa o registo de uma transação financeira associada a uma ou mais reservas. É o componente responsável por consolidar o processo de pagamento, desde a criação da intenção de pagamento na Stripe até à sua confirmação ou eventual cancelamento. O pagamento está sempre vinculado a um utilizador e pode abranger várias reservas pendentes, tornando-se uma peça central na finalização de compras.

Atributos Principais

Campo	Tipo	Descrição
id	id	Identificador único do pagamento.
stripe_payment_intent_id	string	Identificador da intenção de pagamento gerado pela Stripe.
total_price	decimal	Valor total do pagamento. Não pode ser negativo.
status	string	Estado atual do pagamento: "PENDING", "CONFIRMED" ou "CANCELLED".

method_payment	string	Método de pagamento utilizado. Atualmente, apenas "CREDIT_CARD" é suportado.
user_id	id	Referência ao utilizador que efetuou o pagamento.
reservations	lista	Lista de reservas associadas a este pagamento.
inserted_at	datetime	Data de criação do pagamento.
updated_at	datetime	Data da última atualização do pagamento.

Regras de Negócio

- Campos obrigatórios na criação: stripe_payment_intent_id, total_price, status, method_payment e user_id.
- Valor total não pode ser negativo: O campo total_price deve ser igual ou superior a zero.
- Status com valores limitados: Apenas "PENDING", "CONFIRMED" e "CANCELLED" são permitidos.
- Método de pagamento controlado: Apenas "CREDIT_CARD" é aceite no campo method_payment.
- Unicidade do ID de pagamento da Stripe: O stripe_payment_intent_id deve ser único, evitando duplicações.
- Associação com reservas: Um pagamento pode incluir múltiplas reservas, associadas através de uma tabela de junção.
- Criação de intenção de pagamento (Stripe):
    - Utiliza o módulo CreatePaymentIntent.
    - Valida que o valor total seja maior que zero.
    - Converte o valor para centavos antes de enviar à Stripe.
    - Adiciona metadados derivados das reservas.

    - Em caso de falha, erros são capturados e tratados com mensagens descritivas.

- Agrupamento de reservas pendentes:
    - O módulo CreateGroupedPaymentIntent permite consolidar todas as reservas com status "PENDING" de um cliente num único pagamento.
    - Calcula o valor total e cria a intenção de pagamento correspondente.
    - As reservas são atualizadas com o stripe_payment_intent_id.
    - Um registo de pagamento é criado e vinculado às reservas.

- Cancelamento de pagamento:
    - Através do módulo CancelPaymentIntent, permite-se o cancelamento de intenções de pagamento ainda não confirmadas.
    - Garante que o utilizador tem permissão sobre a operação e que o estado do pagamento permite cancelamento.
    - Utiliza a API da Stripe e trata erros detalhadamente.

- Anexação de método de pagamento:
    - O módulo AttachPaymentMethod permite ao utilizador definir um método de pagamento padrão a partir de um token Stripe.
    - Realiza a conversão do token num PaymentMethod, associa-o ao cliente e atualiza as definições da Stripe.

- Confirmação do pagamento:
    - O módulo Confirm atualiza o estado de um pagamento para "CONFIRMED" com base no stripe_payment_intent_id.
    - Garante que o pagamento existe, pertence ao utilizador e está válido para ser confirmado.
    - Após a confirmação, as reservas são automaticamente carregadas.

- Consulta de pagamentos:
    - O módulo Get permite obter a lista de todos os pagamentos realizados por um determinado utilizador, ordenados por data de criação.

Módulos e Funcionalidades

O sistema de pagamentos da plataforma foi desenhado para garantir integração segura e transparente com a Stripe, proporcionando ao utilizador uma experiência fluida e consistente. A arquitetura modular permite a separação clara de responsabilidades, desde a criação e agrupamento de pagamentos até à gestão de métodos de pagamento e tratamento de erros. A entidade Payment é o núcleo desta infraestrutura, ligando reservas, utilizadores e a plataforma de pagamentos externos de forma rastreável e segura.

1. Payments.Create

O módulo Payments.Create é responsável por orquestrar a criação de um pagamento a partir de múltiplas reservas pendentes de um usuário. Seu principal objetivo é garantir que apenas reservas válidas e ainda não pagas sejam consideradas para geração do pagamento. O fluxo inicia com a filtragem das reservation_ids fornecidas, removendo aquelas que já estão associadas a pagamentos existentes. Em seguida, o sistema busca as reservas restantes que pertencem ao usuário, estão com status "PENDING" e são válidas.

Se nenhuma reserva elegível for encontrada, a transação é abortada com um erro claro. Caso contrário, o sistema calcula o valor total do pagamento somando o total de cada reserva e cria o registro do pagamento com status "PENDING" e método de pagamento "CREDIT_CARD". Após isso, as reservas são associadas ao pagamento via tabela intermediária payments_reservations. Toda a operação é realizada dentro de uma transação atômica (Repo.transaction/1), garantindo consistência e rollback em caso de falhas.

2. Payments.Get

O módulo Payments.Get oferece uma interface para listagem de todos os pagamentos realizados por um determinado usuário. Ele executa uma query ordenada por data de criação (inserted_at) de forma decrescente, retornando os registros de forma eficiente e filtrada por user_id.

3. Payments.GetWithReservations

Este módulo amplia a funcionalidade de Payments.Get, permitindo recuperar um pagamento específico juntamente com as reservas associadas a ele. A consulta é baseada no payment_id e no user_id, e inclui preload das associações, garantindo que os dados estejam prontos para exibição detalhada no front-end.

4. Payments.Delete

O módulo Payments.Delete trata da exclusão de registros de pagamento. A operação somente é permitida se o pagamento pertencer ao usuário e não tiver status "CONFIRMED". Caso essas condições não sejam atendidas, a operação retorna erro específico, assegurando a integridade de transações já finalizadas.

5. Payments.ConfirmPayment

O módulo ConfirmPayment realiza a confirmação de uma intenção de pagamento (payment_intent_id) previamente criada via Stripe. O processo verifica a autenticidade do usuário, localiza a intenção via Stripe API e aplica regras de validação específicas. Em caso de sucesso, a confirmação é registrada; caso contrário, mensagens de erro apropriadas são retornadas, incluindo tratamento de falhas da API da Stripe.

6. Payments.CreateGroupedPaymentIntent

Este módulo automatiza a criação de uma nova payment_intent no Stripe com base na soma total das reservas pendentes do usuário. Ele calcula o valor total, gera metadados a partir das reservas e utiliza o Stripe SDK para criar a intenção de pagamento, retornando o payment_intent_id gerado. Isso permite consolidar múltiplas reservas em um único pagamento, otimizando a experiência do usuário.

7. Payments.CancelPaymentIntent

Este módulo gerencia o cancelamento seguro de uma intenção de pagamento (payment_intent) via Stripe. Ele valida se o usuário é o proprietário da intenção, verifica o status atual para evitar cancelamentos indevidos e, se tudo

estiver correto, realiza a operação via Stripe API. Erros são tratados com mensagens claras e apropriadas para o front-end.

8. Payments.AttachPaymentMethod

O módulo AttachPaymentMethod é utilizado para associar um método de pagamento (como cartão de crédito) ao usuário na Stripe, a partir de um token gerado no front-end. Ele cria um PaymentMethod com base no token, o associa ao customer_id do Stripe e define esse método como o padrão para futuras transações. Esse processo é essencial para permitir cobranças automatizadas e facilitar pagamentos futuros.

6. CAMADAS E RESPONSABILIDADES

O backend do WeaveTrip adota uma arquitetura modular baseada nos princípios da Clean Architecture, promovendo uma separação clara entre as responsabilidades das diversas camadas do sistema. Essa estrutura favorece a manutenção, escalabilidade, testabilidade e organização do código.

- **6.1 Resolver (Interface GraphQL)**

  Responsabilidade:

  Os Resolvers representam a camada responsável pela entrada de dados via GraphQL. Cada funcionalidade principal da aplicação (como Reservations, MediaPackages e Users) possui um resolver dedicado. O papel dos resolvers é orquestrar a execução da lógica de negócio, transformando as requisições GraphQL em comandos a serem processados pelos módulos de domínio.

  Comunicação:

  Os resolvers invocam diretamente os delegates de cada módulo de domínio (WeaveTrip.Reservations, WeaveTrip.MediaPackages, entre outros), que centralizam a interface pública de cada funcionalidade.

- **6.2 Context (Domínio de Negócio)**

  Responsabilidade:

  A camada de Context corresponde ao núcleo da aplicação, onde está concentrada a lógica de negócio. Cada módulo do domínio (Reservations, MediaPackages, Users) é autônomo e modular, contendo arquivos específicos para cada operação CRUD (como create.ex, get.ex, update.ex, delete.ex). Cada módulo possui ainda um delegate (por exemplo, reservation.ex), que funciona como uma interface pública, agrupando e expondo as funções de forma coesa e organizada.

  Comunicação:

  Os resolvers chamam os delegates da camada de contexto, que, por sua vez, podem coordenar operações mais complexas, integrando diversas funções internas ou outros módulos. A camada de contexto se comunica com a infraestrutura de persistência por meio do Ecto.Repo.

- **6.3 Schema (Definição GraphQL)**

  Responsabilidade:

  O schema GraphQL centraliza a definição da interface pública da API, estabelecendo os tipos de dados, mutations e queries disponibilizados aos clientes. Todas as operações expostas pelo sistema são mapeadas neste schema, que define como os resolvers devem ser acionados e qual será a estrutura das respostas.

  Comunicação:

  O schema conecta cada operação a um resolver específico, funcionando como um roteador entre o cliente e a aplicação.

- **6.4 Controller (Interface REST — opcional)**

  Responsabilidade:

  Embora o foco principal da aplicação seja a interface GraphQL, a arquitetura prevê o uso de Controllers como camada de entrada para endpoints REST, se necessário. Os controllers seguem o padrão tradicional

do Phoenix, lidando com requisições HTTP, processando parâmetros e delegando a execução das operações para os módulos de contexto.

Comunicação:

Assim como os resolvers, os controllers chamam os delegates dos módulos de domínio, possibilitando o reuso da lógica central da aplicação, sem duplicação de código.

- 6.5 **Service (Lógica Auxiliar e Transversal)**

Responsabilidade:

Embora não formalizada como uma camada específica, as funções que realizam tarefas transversais — como autenticação, autorização, envio de e-mails ou notificações — podem ser encapsuladas em Services ou Helpers. Esses serviços são utilizados tanto pela camada de domínio quanto pelas interfaces (REST ou GraphQL), conforme a necessidade.

Comunicação:

Os módulos de domínio ou middleware chamam esses serviços diretamente para realizar operações que não fazem parte da lógica central de negócio, mas são essenciais para o funcionamento do sistema.

Comunicação entre as Camadas

O fluxo de comunicação entre as camadas segue o padrão de entrada → domínio → persistência, com cada camada focada em um nível específico de abstração e responsabilidade:

1. **Cliente → Schema → Resolver:**
   O cliente realiza uma requisição (query ou mutation), que é interpretada pelo schema GraphQL e direcionada ao resolver correspondente.
2. **Resolver → Context (Delegate):**
   O resolver invoca o delegate do módulo de domínio, passando os dados validados para a execução da lógica de negócio.
3. **Context → Infraestrutura (Repo):**
   O módulo de domínio executa a lógica, valida as regras de negócio e, se necessário, realiza operações de persistência de dados via WeaveTrip.Repo.
4. **Context → Resolver → Cliente:**
   O resultado da operação é retornado do domínio ao resolver, que formata a resposta conforme definido no schema GraphQL e envia ao cliente.

7. GESTÃO DE CONFIGURAÇÕES E AMBIENTES

No contexto da aplicação WeaveTrip, a configuração é realizada utilizando os arquivos config/config.exs, config/dev.exs, config/prod.exs e config/runtime.exs, seguindo as boas práticas recomendadas pela comunidade Elixir e Phoenix. O arquivo config.exs centraliza as definições comuns a todos os ambientes, enquanto os arquivos específicos (dev.exs, test.exs, prod.exs) contêm configurações ajustadas conforme o ambiente de execução. Por exemplo, no ambiente de desenvolvimento (dev.exs), são habilitadas funcionalidades como code_reloader e debug_errors, que facilitam a depuração, mas são desativadas em produção para garantir segurança e desempenho.

Além disso, a aplicação diferencia claramente os ambientes desenvolvimento (dev), teste (test) e produção (prod), utilizando o Mix.env() e o comando import_config "#{config_env()}.exs" para importar dinamicamente a configuração correspondente. Essa prática assegura que cada ambiente opere com parâmetros adequados, como diferentes conexões de banco de dados, chaves de segurança e endpoints.

Outro aspecto importante é o uso de variáveis de ambiente para proteger informações sensíveis, como senhas, chaves secretas e URLs de banco de dados. A aplicação faz uso da função System.get_env/2 para obter esses valores, evitando que dados críticos sejam expostos no controle de versão.

Embora a utilização de valores padrão (|| "valor") possa ser útil em ambientes de desenvolvimento, recomenda-se evitá-los em ambientes produtivos para garantir a segurança da aplicação. Para reforçar essa prática, o projeto inclui a dependência dotenv, que permite o carregamento automático de variáveis definidas em arquivos .env, facilitando a configuração local sem comprometer a segurança.

Além disso, configurações que impactam diretamente a operação do sistema, como as credenciais do ExAws para integração com o MinIO e parâmetros de inicialização do Phoenix Endpoint, são adequadamente parametrizadas. A separação dessas informações promove um ambiente mais seguro e facilita o processo de implantação, especialmente quando se utiliza serviços de integração contínua (CI/CD) ou plataformas de nuvem.

A configuração do runtime, efetuada no arquivo runtime.exs, permite que variáveis críticas sejam carregadas dinamicamente durante a execução da

aplicação, conforme as variáveis de ambiente definidas no sistema operacional ou no ambiente de orquestração (por exemplo, Docker, Kubernetes ou serviços de hospedagem como Gigalixir ou Fly.io). Esta abordagem elimina a necessidade de recompilar a aplicação para alterações de configuração, reduzindo riscos e agilizando processos de manutenção.

8. SEGURANÇA

A aplicação implementa um robusto sistema de segurança que abrange autenticação, autorização, criptografia de senhas e validação de entradas, protegendo contra diversas vulnerabilidades conhecidas.

Autenticação

A autenticação é realizada através do uso do Guardian, uma biblioteca consolidada no ecossistema Elixir para autenticação baseada em tokens JWT (JSON Web Tokens). O módulo WeaveTrip.Guardian define a lógica para geração e verificação dos tokens. A função subject_for_token/2 utiliza o user.id como identificador único no token, enquanto resource_from_claims/1 recupera o usuário a partir das claims do token.

O processo de autenticação via credenciais (email e senha) é implementado no método authenticate/2, que busca o usuário pelo email e valida a senha utilizando o Pbkdf2.verify_pass/2, garantindo segurança no armazenamento e validação das senhas através de uma função de derivação de chave robusta e resistente a ataques de força bruta.

Autorização

A autorização é implementada através de middlewares personalizados utilizando Absinthe, framework GraphQL para Elixir. O módulo WeaveTripWeb.Middleware.Authenticate verifica se o current_user está presente no contexto da resolução. Caso contrário, a requisição é bloqueada com erro "Unauthorized".

Já o módulo WeaveTripWeb.Middleware.CheckRole realiza a verificação de permissões baseadas no papel (role) do usuário, permitindo apenas que funções específicas sejam executadas por determinados perfis, como admin, user ou agency. A checagem é feita de maneira dinâmica, garantindo flexibilidade e clareza

nas mensagens de erro. Essa abordagem evita acessos indevidos a recursos sensíveis da aplicação.

Criptografia de Senhas

O sistema emprega Pbkdf2 para a criptografia das senhas, uma prática essencial para a proteção dos dados sensíveis dos usuários. O algoritmo adiciona um fator de dificuldade computacional ao processo de verificação, tornando ataques de dicionário e força bruta impraticáveis. Durante o login, a senha fornecida é comparada com o hash armazenado através do Pbkdf2.verify_pass/2.

Validações e Proteção contra Vulnerabilidades

A aplicação também implementa proteções importantes contra vulnerabilidades comuns:

- O módulo WeaveTripWeb.Plugs.SafeParsers encapsula a execução de Plug.Parsers, tratando especificamente erros como RequestTooLargeError (HTTP 413), evitando que requisições excessivamente grandes sobrecarreguem ou causem falhas no servidor.

- A estrutura de autenticação e autorização garante que apenas usuários autenticados e com as devidas permissões possam acessar ou modificar recursos, prevenindo ataques como escalada de privilégios.

- O WeaveTripWeb.AuthPipeline realiza a verificação automática dos tokens JWT enviados no cabeçalho das requisições, carregando o recurso associado ao token e atribuindo-o de forma segura ao conn.assigns[:current_user] com o plug assign_current_user. Esse padrão evita a necessidade de repetidas consultas e centraliza o controle de autenticação.

Além disso, práticas como o tratamento adequado de erros e a separação clara de responsabilidades entre autenticação, autorização e parsing de requisições reforçam a segurança geral da aplicação.

9. TRATAMENTO DE ERROS E LOGS

O sistema de tratamento de erros da API WeaveTrip foi desenvolvido para fornecer respostas claras, padronizadas e seguras para os clientes da API. Além disso, oferece suporte à rastreabilidade por meio de logs estruturados, facilitando o monitoramento e a depuração de falhas.

A API segue as diretrizes do Absinthe (framework GraphQL utilizado no projeto) para formatar os erros, incluindo informações úteis como mensagens descritivas, códigos simbólicos, status HTTP e a localização exata do erro na operação. Os erros retornados seguem o seguinte padrão:

Campo	Descrição
message	Mensagem amigável explicando o erro ocorrido.
extensions.code	Código simbólico que representa o tipo de erro (ex.: UNAUTHORIZED).
extensions.http_status	Código de status HTTP equivalente (ex.: 401, 403, 422, 500).
path	Campo da operação GraphQL onde o erro ocorreu.

Observações

Erros de validação, como formato de e-mail inválido ou senha muito curta, retornam mensagens detalhadas geradas automaticamente pelos Ecto.Changesets, informando quais campos são inválidos e o motivo. A aplicação utiliza o módulo nativo Logger do Elixir para registro e rastreamento de eventos importantes. Os logs são classificados de acordo com o nível de severidade, permitindo filtragem eficiente por ferramentas externas de monitoramento e análise.

Níveis de Log Utilizados

Nível	Utilização
:info	Registra acessos, autenticações e ações bem-sucedidas.
:warn	Registra erros de negócio, inconsistências ou uso incorreto da API.
:error	Registra falhas inesperadas, como exceções e problemas críticos.

10. BANCO DE DADOS E MODELOS DE DADOS

O banco de dados da API WeaveTrip foi projetado com base no modelo relacional, utilizando o PostgreSQL como sistema gerenciador. Sua estrutura foi desenvolvida e versionada por meio do Ecto, biblioteca oficial do ecossistema Elixir, o que garante consistência, rastreabilidade de mudanças e integridade dos dados durante o desenvolvimento e a evolução da aplicação. As alterações no esquema são controladas por meio de migrations, permitindo a aplicação segura e ordenada de mudanças estruturais ao longo do tempo.

A modelagem de dados foi construída de forma a representar fielmente os principais fluxos de negócio da aplicação, como a criação e administração de pacotes de viagem por agências, a realização de reservas por clientes, e o controle de disponibilidade e mídia associada aos pacotes. O banco de dados é composto por entidades principais como users, travel_packages, reservations, media_packages e payments, cada uma com atributos específicos e relacionamentos que asseguram a integridade referencial entre as informações armazenadas.

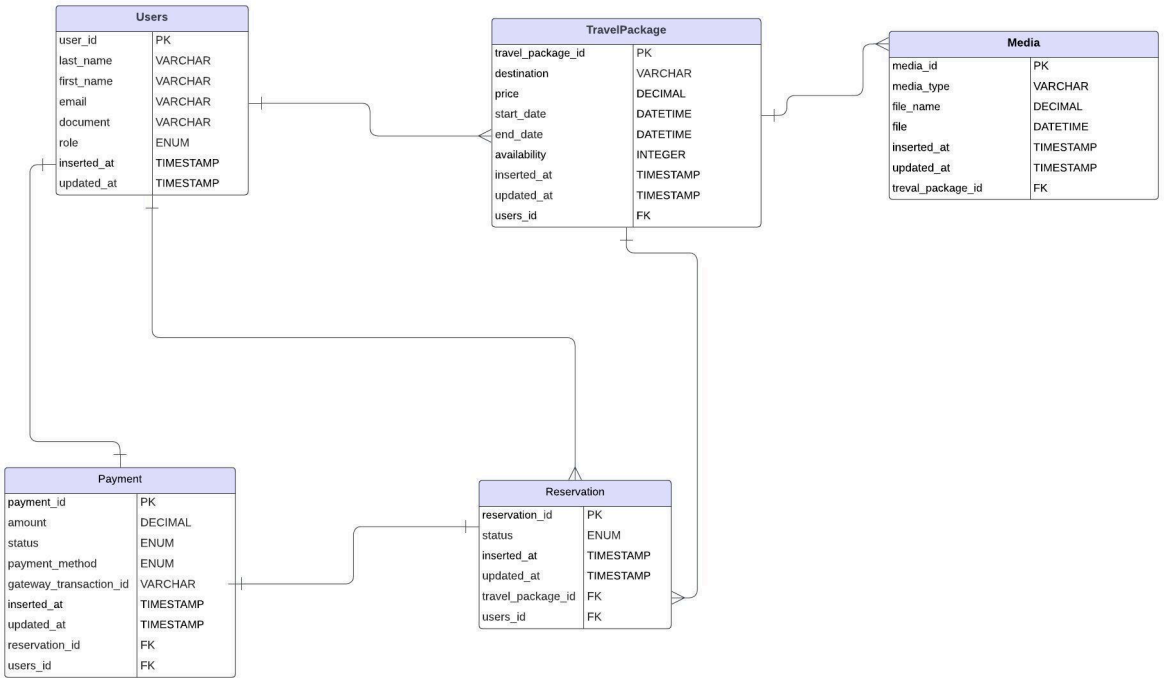

Figura 2 - Diagrama de Banco de Dados - Weave Trip

A tabela users armazena os dados de todos os usuários da plataforma, que podem assumir diferentes perfis de uso — como clientes, agências de viagem ou administradores do sistema. Para isso, o campo role foi introduzido como discriminador funcional. Cada usuário possui atributos como nome, e-mail, documento (CPF ou CNPJ), senha (armazenada de forma segura com hash), telefone, data de nascimento e endereço completo, incluindo CEP, rua, bairro, número, cidade e estado. Essa tabela também é responsável por centralizar os relacionamentos com outras entidades, como pacotes criados (no caso de agências), reservas efetuadas e, futuramente, pagamentos realizados.

A entidade travel_packages representa os pacotes turísticos cadastrados pelas agências. Cada pacote contém informações como nome, destino, descrição detalhada, preço por pessoa, número total de vagas disponíveis, datas de início e fim da viagem, status (ativo ou inativo), além da política de cancelamento adotada pela agência. A agência responsável é identificada pelo campo user_id, que referencia diretamente o usuário criador do pacote. Essa entidade também estabelece relacionamento com a tabela de reservas e com a tabela de mídias, permitindo a associação de imagens ilustrativas ou promocionais a cada pacote.

Já a tabela reservations é responsável por armazenar os dados das reservas realizadas por clientes. Cada reserva associa um cliente, uma agência e um pacote de viagem, permitindo o controle completo da transação. Os principais atributos incluem a data da reserva, uma data de expiração (que define a validade da reserva caso o pagamento não seja realizado), o número de viajantes, o método de pagamento selecionado e o valor total calculado com base na quantidade de pessoas.

A reserva também possui um status que pode assumir os valores "pending", "confirmed", "canceled" ou "expired", possibilitando o rastreamento do estado da reserva ao longo do tempo. Um token UUID único é gerado para cada reserva como forma de identificação segura e rastreável. A política de cancelamento vigente no momento da reserva também é armazenada, garantindo que futuras alterações não afetem reservas já realizadas.

Complementando o conjunto de dados, a entidade media_packages permite associar arquivos de mídia aos pacotes de viagem. Esses arquivos podem incluir imagens e vídeos promocionais que ilustram melhor o conteúdo do pacote para os usuários da plataforma. Os atributos dessa tabela incluem a URL do arquivo, uma legenda descritiva, um identificador que define se a mídia é a principal (para fins de destaque na interface), e timestamps automáticos. Essa relação com os pacotes de viagem segue um modelo 1:N, no qual cada pacote pode ter múltiplos arquivos de mídia vinculados.

Importante destacar que a entidade payments já foi concebida em nível de modelagem, embora ainda não esteja implementada na aplicação atual. Esta tabela será responsável por registrar os pagamentos efetuados pelos usuários referentes às reservas confirmadas. Ela incluirá campos como valor, status, método de pagamento, e identificadores de transações, principalmente para integração com gateways de pagamento externos. O relacionamento previsto estabelece uma associação direta com as tabelas users e reservations, garantindo rastreabilidade e transparência nas transações financeiras. A implementação dessa funcionalidade está planejada para uma versão futura do sistema.

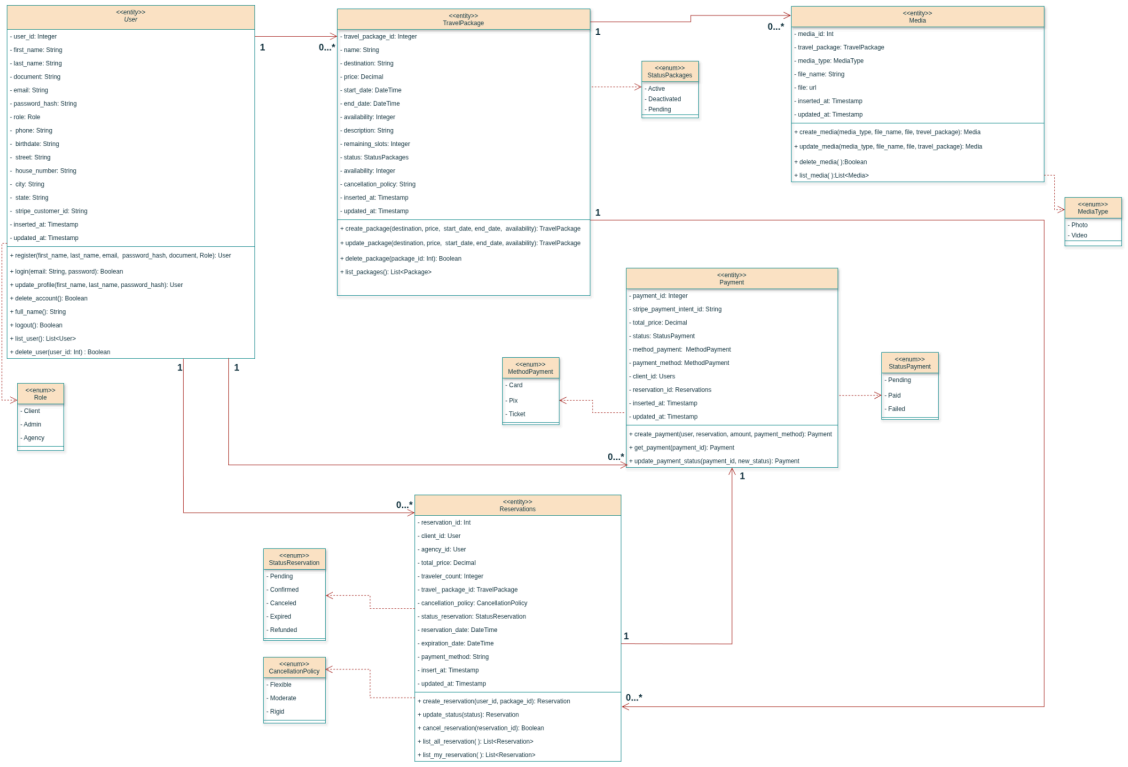

Figura 3 - Diagrama de Classes UML - Weave Trip

11. ENDPOINTS E API CONTRACT

A API do WeaveTrip é baseada em GraphQL e está acessível por meio de uma única URL de entrada, que serve tanto para queries quanto para mutations. Toda a comunicação entre o cliente e o servidor ocorre via requisições HTTP POST contendo o payload em formato JSON, respeitando a estrutura do GraphQL.

Estrutura da URL de Acesso:

- **Endpoint principal (produção):**
- **Endpoint local (desenvolvimento):** http://localhost:4000/api/graphql

A API utiliza autenticação via tokens JWT, que devem ser enviados no cabeçalho das requisições autenticadas: *Authorization: Bearer <token>*. As operações que exigem autenticação (como visualização de dados do usuário logado, criação de reservas, etc.) verificarão a validade e a permissão do token.

Versionamento

Atualmente, a API não adota versionamento explícito por URL ou headers, mas a estrutura do GraphQL permite a evolução do schema sem impactar funcionalidades existentes, seguindo as boas práticas de compatibilidade retroativa. Mudanças significativas serão documentadas em changelogs e refletidas na documentação oficial.

Queries disponíveis

As queries permitem buscar dados de diferentes entidades do sistema, como usuários, pacotes de viagem e reservas. Exemplos:

1. me: retorna os dados do usuário logado.
2. getUser(id: ID!): busca um usuário pelo ID.
3. listUsers: lista todos os usuários.
4. getTravelPackage(id: ID!): busca um pacote de viagem específico.
5. listTravelPackages: lista todos os pacotes de viagem.
6. getReservation(id: ID!): retorna uma reserva específica.
7. listReservations: lista todas as reservas.
8. myReservations: lista as reservas do usuário logado.

Mutations disponíveis

As mutations permitem executar operações que modificam o estado do sistema. Exemplos:

1. signUp(input: SignUpInput!): cadastro de um novo usuário.
2. signIn(email: String!, password: String!): login e obtenção de token JWT.
3. signOut: logout do usuário.
4. updateUser(input: UpdateUserInput!): atualização de dados do usuário.
5. changePassword(input: ChangePasswordInput!): alteração de senha.
6. createTravelPackage(input: TravelPackageInput!): criação de pacote (admin).
7. updateTravelPackage(input: TravelPackageInput!): atualização de pacote.
8. deleteTravelPackage(id: ID!): remoção de pacote.
9. createReservation(input: ReservationInput!): criação de reserva.
10. updateReservation(input: ReservationInput!): atualização de reserva.
11. confirmReservation(id: ID!): confirmação de uma reserva.
12. cancelReservation(id: ID!): cancelamento de uma reserva.
13. deleteReservation(id: ID!): exclusão de reserva.

12. MANUAL DE INSTALAÇÃO E USO

O **WeaveTrip** é uma plataforma de gestão de pacotes turísticos voltada para agências de viagem e usuários finais, desenvolvida em Elixir com o framework Phoenix. O sistema permite o cadastro, gerenciamento e reserva de pacotes de viagem, com controle de vagas, políticas de cancelamento, pagamentos e outras funcionalidades administrativas.

Este manual tem como objetivo fornecer instruções detalhadas para:

- Instalar e executar o projeto em ambiente local (desenvolvimento);
- Subir o sistema em produção usando containers Docker;
- Compilar, rodar, parar e reiniciar o sistema;
- Gerenciar serviços auxiliares como banco de dados e armazenamento de mídias.

12.1. Tecnologias Utilizadas

- Linguagem: Elixir 1.18.2 (compilado com Erlang/OTP 27)
- Framework Web: Phoenix 1.7.20
- API: GraphQL via Absinthe (~> 1.7)
- Banco de Dados: PostgreSQL (>=13)
- Armazenamento de mídias: MinIO (compatível com S3)
- Gerenciador de pacotes frontend: Node.js (>=16)
- Orquestração de containers: Docker e Docker Compose

12.2. Instalação em Ambiente Local

1. Clonando o Repositório

```
git clone https://github.com/usuario/weavetrip.git
cd weavetrip
```

2. Instalação das Dependências Elixir

- Certifique-se de que Elixir e Erlang estão instalados. Em seguida, execute:

```
mix deps.get
```

3. Instalação das Dependências Frontend

```
cd assets
npm install
cd ..
```

4. Configuração do Banco de Dados

- Configure o arquivo config/dev.exs com as credenciais do PostgreSQL local.
- Execute os seguintes comandos para preparar o banco:

```
mix ecto.create
mix ecto.migrate
```

5. Variáveis de Ambiente

- Crie um arquivo .env com os seguintes parâmetros:

```
DATABASE_URL=postgres://postgres:postgres@localhost/weavetrip_dev
SECRET_KEY_BASE=CHAVE_GERADA
```

MINIO_ACCESS_KEY=minioadmin
MINIO_SECRET_KEY=minioadmin
MINIO_HOST=http://localhost:9000

- Carregue as variáveis:
  source .env

- A chave SECRET_KEY_BASE pode ser gerada com o comando:
  mix phx.gen.secret

6. Executando a Aplicação
   mix phx.server

A aplicação estará disponível em http://localhost:4000.

12.3. Instalação com Docker

1. Configuração Inicial
   - Crie um arquivo .env com as variáveis necessárias (conforme seção anterior).
   - Certifique-se também de possuir um docker-compose.yml corretamente configurado, incluindo serviços para:
     - app (aplicação Elixir/Phoenix)
     - db (PostgreSQL)
     - minio (armazenamento de mídias)

2. Build e Execução dos Containers
   docker-compose up --build

3. Preparação do Banco de Dados
   - Após a aplicação estar em execução, execute:

   docker-compose exec app mix ecto.create
   docker-compose exec app mix ecto.migrate
   docker-compose exec app mix run priv/repo/seeds.exs

4.4. Parar os Containers
   docker-compose down

12.4. Deploy em Servidor

1. Instalação do Docker e Docker Compose
   - Acesse o servidor via SSH e execute os comandos:

     ```
     sudo apt update
     sudo apt install -y docker.io docker-compose
     sudo systemctl enable docker
     sudo systemctl start docker
     ```

   - Verifique as versões:

     ```
     docker --version
     docker-compose --version
     ```

2. Clonando o Repositório
   - No diretório de sua preferência:

     ```
     git clone https://github.com/usuario/weavetrip.git
     cd weavetrip
     ```

3. Criação do Arquivo .env (Produção)

   - Crie um arquivo .env na raiz do projeto com variáveis seguras:

     ```
     touch .env
     ```

Conteúdo exemplo:

```
DATABASE_URL=ecto://postgres:postgres@db:5432/weavetrip_prod
SECRET_KEY_BASE=<CHAVE_GERADA>
MINIO_ACCESS_KEY=minioadmin
MINIO_SECRET_KEY=minioadmin
MINIO_HOST=http://minio:9000
```

   - A chave SECRET_KEY_BASE pode ser gerada com:

     ```
     mix phx.gen.secret
     ```

Atenção: o arquivo .env **não deve ser versionado** (adicione ao .gitignore) e deve ter permissões restritas:

```
chmod 600 .env
```

4. Subida dos Containers

- Inicie o sistema em modo desacoplado (em segundo plano):

```
docker-compose up -d --build
```

5. Migração e Seed do Banco de Dados

- Após os containers estarem ativos, execute:

```
docker-compose exec app mix ecto.create
docker-compose exec app mix ecto.migrate
docker-compose exec app mix run priv/repo/seeds.exs
```

6. Persistência de Volumes

- Certifique-se de que o docker-compose.yml utiliza volumes persistentes para:
  - Banco de dados PostgreSQL;
  - Armazenamento de objetos do MinIO.

Exemplo de definição no docker-compose.yml:

```
volumes:
  db_data:
  minio_data:
```

Esses volumes são fundamentais para garantir que os dados não sejam perdidos ao reiniciar ou reconstruir os containers.

12.5. Gerenciamento de Logs e Manutenção

1. Visualizar logs

- Logs gerais do Docker:

```
docker-compose logs -f
```

- Logs específicos do app:

```
docker-compose logs -f app
```

2. Atualizar imagens e reiniciar containers

- Para atualizar a aplicação:

```
docker-compose pull
docker-compose up -d --build
```

3. Reiniciar apenas a aplicação

```
docker-compose restart app
```

4. Backup

Para realizar backups:

- Banco de dados PostgreSQL:
  - Utilize pg_dump dentro do container ou conectando externamente.
- Arquivos do MinIO:
  - Copie o volume minio_data ou configure sincronização com outra instância compatível com S3.

- Exemplo de backup do banco (usando o host):

```
docker exec -t weavetrip-db pg_dump -U postgres weavetrip_prod > backup.sql
```

12.6. Boas Práticas em Produção

- Segurança de variáveis: o .env nunca deve ser versionado. Restrinja o acesso.
- SECRET_KEY_BASE: gere uma chave segura e única para produção.
- TLS: sempre configure HTTPS com certificado válido.
- Volumetria: use volumes persistentes para o banco de dados e armazenamento.
- Backups regulares: automatize o backup dos volumes e banco.
- Atualizações: mantenha o sistema, dependências e containers atualizados.

- Monitoramento: utilize ferramentas de monitoramento e alerta, como UptimeRobot, Prometheus ou soluções customizadas.
- Logs externos: configure agregadores de logs (ex: Loki, Logstash ou Fluentd) se necessário.

13. CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O desenvolvimento da API backend do projeto WeaveTrip representou um avanço significativo na consolidação de uma infraestrutura robusta e escalável para gestão de pacotes de viagem. A arquitetura baseada em GraphQL, combinada com autenticação JWT e um conjunto bem definido de regras de negócio, permitiu a criação de um sistema coeso, seguro e orientado a diferentes perfis de usuários (clientes, agências e administradores).

A criação de um sistema de busca avançada e recomendações de pacotes de viagem visa melhorar a experiência do usuário, tornando a navegação mais eficiente e personalizada. O sistema permitirá que os usuários filtrem pacotes por critérios como destino, faixa de preço, datas disponíveis e número de vagas restantes. Além disso, pretende-se incorporar mecanismos de recomendação com base em dados comportamentais e históricos de navegação e reservas. A aplicação dessas técnicas pode contribuir para sugerir pacotes relevantes a cada perfil de usuário, aumentando o engajamento e a conversão dentro da plataforma. Esta funcionalidade será projetada de forma escalável e modular, facilitando sua evolução com base na coleta de dados e feedback dos usuários.

14. REFERÊNCIAS

ABSINTHE. *Absinthe GraphQL – The Elixir GraphQL Toolkit*. [S.l.]: Plataformatec, 2023. Disponível em: https://hexdocs.pm/absinthe. Acesso em: 25 maio 2025.

ELIXIR LANG. *Elixir – Getting Started and Documentation*. [S.l.], 2023. Disponível em: https://elixir-lang.org. Acesso em: 25 maio 2025.

FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Tese (Doutorado) – University of California, Irvine.

GRAPHQL FOUNDATION. *GraphQL Specification*. 2022. Disponível em: https://spec.graphql.org. Acesso em: 25 maio 2025.

MARTIN, Robert C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Prentice Hall, 2017.

PHOENIX FRAMEWORK. *Phoenix – Productive. Reliable. Fast.* [S.l.], 2023. Disponível em: https://www.phoenixframework.org. Acesso em: 25 maio 2025.

TROYES, Saša Jurić. *Elixir in Action*. 2. ed. Shelter Island: Manning Publications, 2019.

CAMPUS CODE. *Elixir: testes unitários e DocTest*. Disponível em: https://campuscode.com.br/conteudos/elixir-testes-unitarios-e-doctest. Acesso em: 30 maio 2025.

ELIXIR SCHOOL. *Testes*. Disponível em: https://elixirschool.com/pt/lessons/basics/testing. Acesso em: 30 maio 2025.

HENRIQUE, True. *Elixir ExUnit: asserções e mocks para chamadas de funções*. Medium, 2021. Disponível em: https://medium.com/true-henrique/elixir-exunit-asser%C3%A7%C3%B5es-e-mocks-parachamadas-de-fun%C3%A7%C3%B5es-f739de6d31a1. Acesso em: 30 maio 2025.

OLIVEIRA, Eduardo Felipe da Silva. *Estudo sobre o uso do Elixir para o desenvolvimento de aplicações web modernas*. 2021. Trabalho de Conclusão de Curso (Graduação em Engenharia de Software) – Universidade Tecnológica Federal do Paraná, Cornélio Procópio, 2021. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/10771. Acesso em: 30 maio 2025.

SOUZA, Paulo. *Meu hello world em Elixir*. Café com Elixir, 2022. Disponível em: https://aprenda.cafecomelixir.com.br/primeiros-passos-em-elixir/meu-hello-world. Acesso em: 30 maio 2025.