Documentação do Projeto

Este projeto é uma API com a funcionalidade de gerenciar tabelas no Supabase, destinada a ser utilizada em um e-commerce. Sua função principal é gerenciar usuários (clientes), produtos e pedidos. Para o desenvolvimento, foram utilizadas as seguintes tecnologias: Fastify, Zod, integração de Type Providers entre Fastify e Zod, Swagger, Prisma e Supabase.
No Supabase, foram configuradas políticas de Row-Level Security (RLS) e criadas algumas views que utilizam triggers para facilitar consultas mais complexas.

Para executar o projeto basta seguir as seguintes instruções

Pré-requisitos

- Node.js (versão 18 ou superior)
- Conta no Supabase
- Conta no Resend (para envio de emails)

Passos para execução

2. Instale as dependências:
   npm install

3. Configure as variáveis de ambiente:
   - Preencha as variáveis conforme o arquivo env.ts

4. Configure o banco de dados no Supabase:
   - Crie as tabelas conforme a estrutura apresentada
   - Execute as políticas RLS documentadas
   - Crie as triggers e views

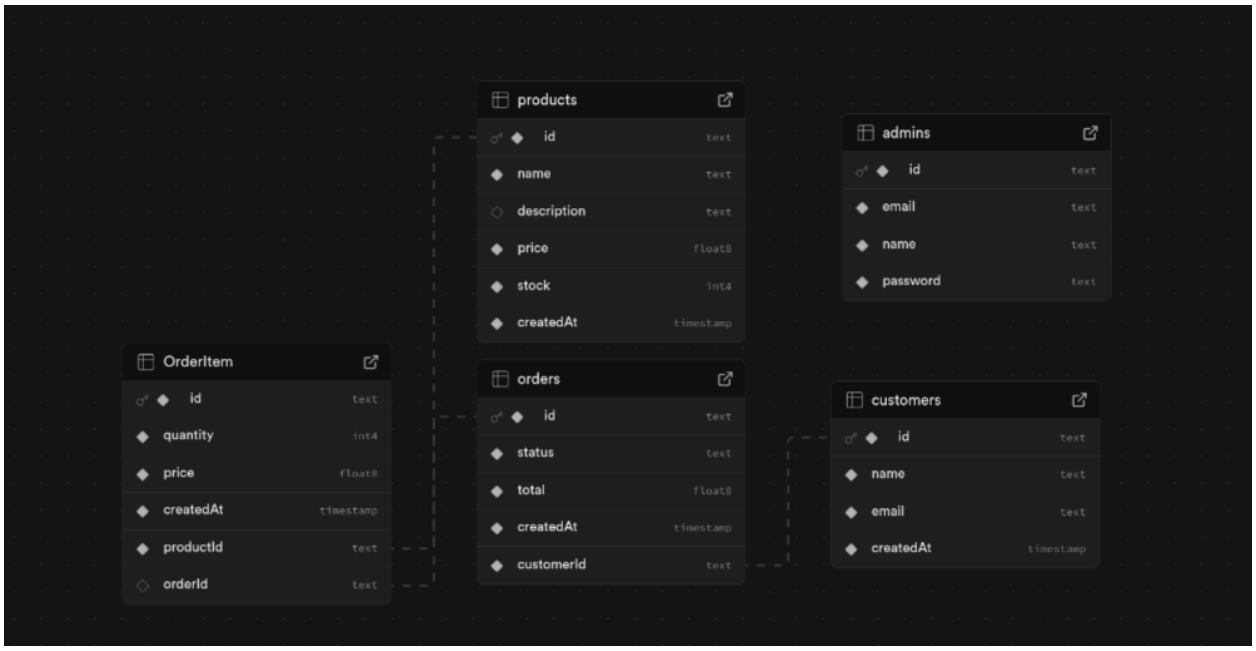
5. Execute as migrações do Prisma:
   npx prisma generate
   npx prisma db push

6. Inicie o servidor:
   npm run dev

7. Acesse a documentação da API:
   http://localhost:3333/docs

Estrutura do Banco de Dados

O banco é composto pelas seguintes tabelas: customers, products, orders, orderItems e admins. A imagem a seguir apresenta o modelo visual do banco utilizando a ferramenta do Supabase:

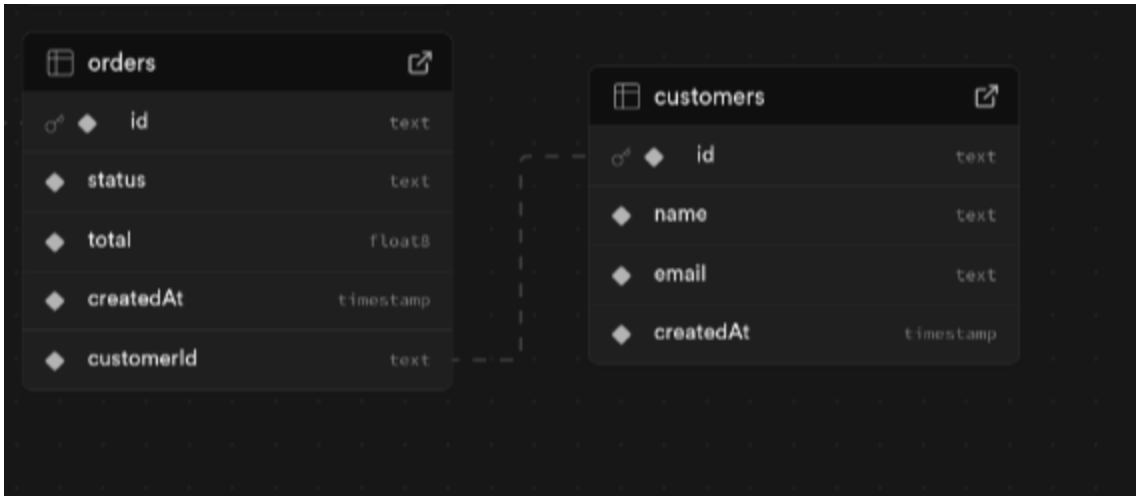


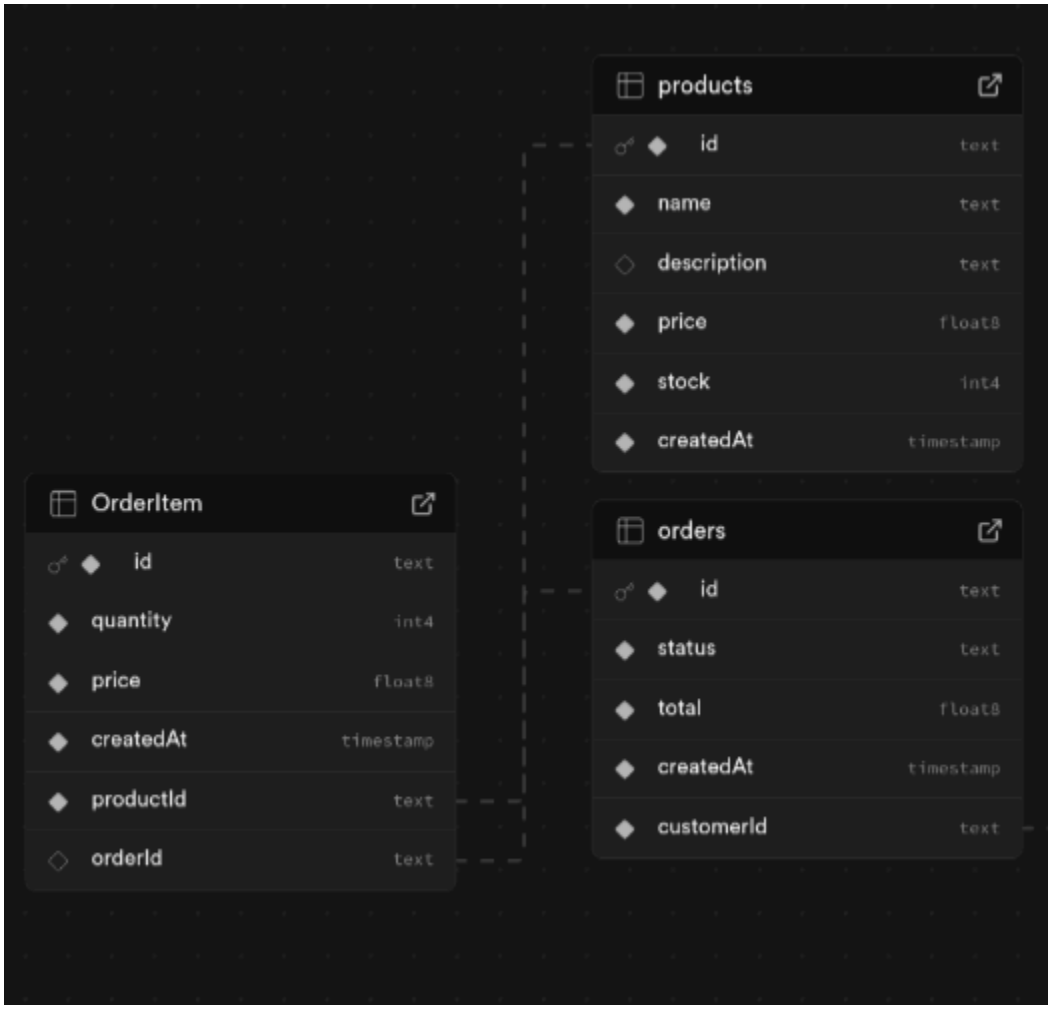
A tabela customers está relacionada à tabela "orders", representando os clientes que efetuaram pedidos, embora alguns clientes possam ainda não ter feito pedidos.

A tabela "orders" está relacionada com orderItems, que detalham os itens que compõem cada pedido. A orderItems por sua vez está relacionada à tabela products, pois é composta pelos produtos que fazem parte do pedido.

A coluna id em todas as tabelas é do tipo text. Entretanto, como o Prisma é utilizado para criação de novos registros, ele padroniza essa coluna no formato UUID, o que é coerente com o requisito do campo id.

O banco foi criado no Supabase conforme as imagens referenciadas, utilizando um arquivo .env, cuja estrutura pode ser verificada através do arquivo env.ts. Esse arquivo também pode servir como exemplo para configuração correta do ambiente.





Testes de API

A documentação da API está disponível no endpoint /docs, que permite testar as rotas via opção "Try Out". Os usuários devem ser criados inicialmente (clientes) e depois realizar login. A função de login retorna um JWT que deve ser utilizado na parte superior da página, no campo de autorização ("Authorize"), possibilitando o teste das rotas que requerem autenticação.

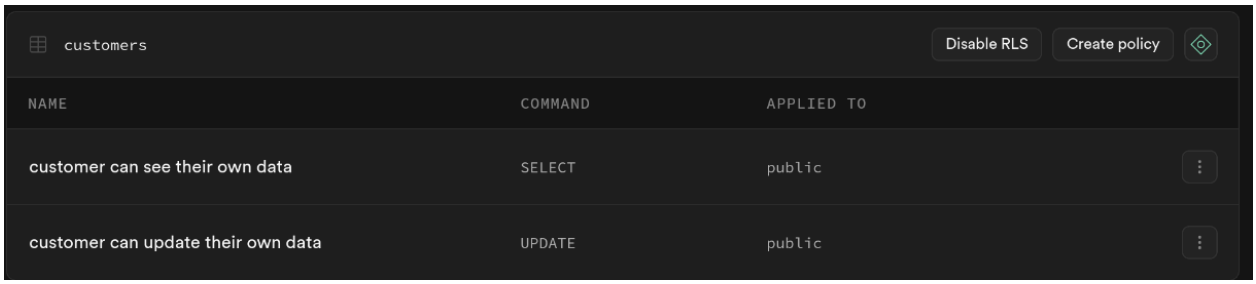
Informações Importantes

O usuário administrador (admin) deve ser criado inteiramente dentro do Supabase, na tabela admins, e também registrado na tabela de autenticação para que possa realizar login e executar a ação de criação de produtos.

Políticas de Row-Level Security (RLS)

Para aumentar a segurança do banco, foram implementadas várias políticas RLS, conforme descrito abaixo:

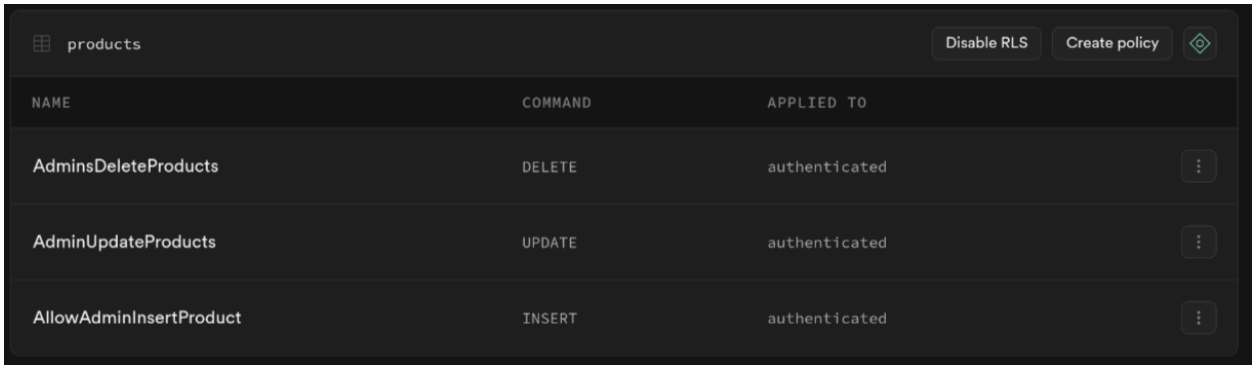
Na tabela customers:




```
USING (
  "id" = auth.uid()::text
  OR EXISTS (SELECT 1 FROM "admins" WHERE "id" = auth.uid()::text)
)
```

Essa política permite que apenas o próprio usuário possa visualizar ou modificar suas informações pessoais, ou que um administrador tenha esse acesso.

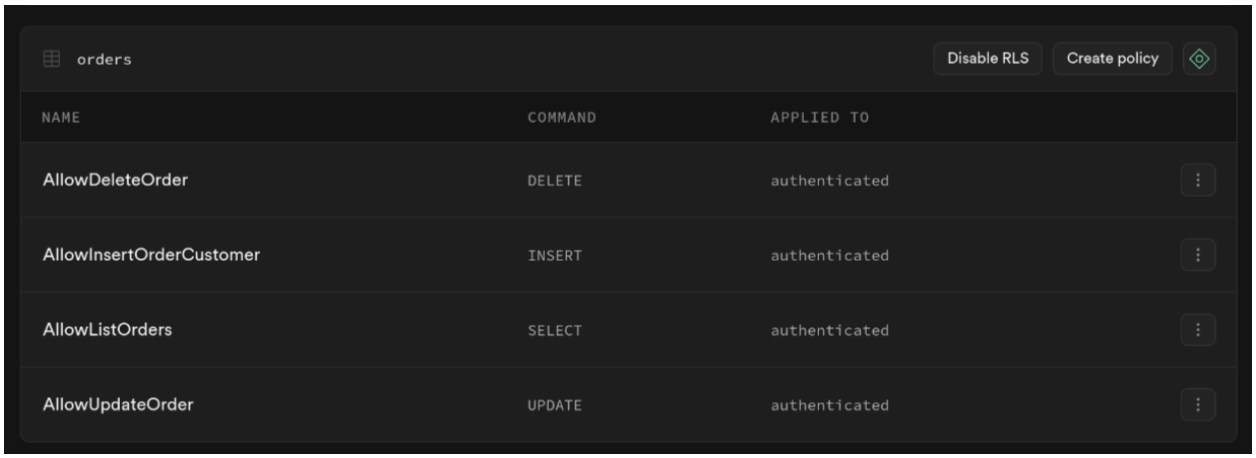
Na tabela Products:




```
USING (
  EXISTS (SELECT 1 FROM "admins" WHERE "id" = auth.uid()::text)
)
```

Somente administradores têm permissão para alterar os dados de produtos, seja inserindo, modificando ou deletando.

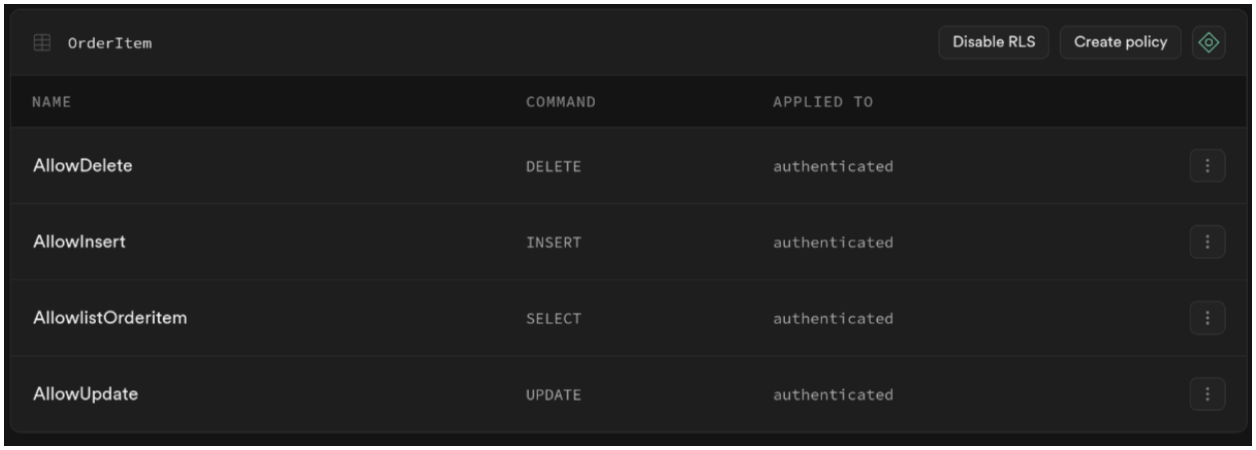
Na tabela orders :




```
USING (
  "customerId" = auth.uid()::text
  OR EXISTS (
    SELECT 1 FROM "admins" WHERE "id" = auth.uid()::text
  )
)
```

Clientes autenticados podem criar pedidos, porém somente o dono do pedido ou um administrador podem visualizar, editar ou deletar os pedidos.

Na tabela OrderItems temos 4 RLS:




```
USING (
  EXISTS (
    SELECT 1 FROM "orders"
    WHERE "orders"."id" = "OrderItem"."orderId"
      AND (
        "orders"."customerId" = auth.uid()::text
        OR EXISTS (SELECT 1 FROM "admins" WHERE "id" = auth.uid()::text)
      )
  )
)
```

Clientes autenticados podem criar pedidos, porém somente o dono do pedido ou um administrador podem visualizar, editar ou deletar os pedidos.

Triggers

Foram configuradas algumas triggers importantes no banco:

1. Trigger para o envio automático de e-mails quando uma conta é criada. Para facilitar o desenvolvimento, no código de teste o campo que indica se a conta está verificada está definido como true.
2. Trigger para preencher automaticamente o valor do campo price em OrderItem. Segue a trigger:

```
CREATE OR REPLACE FUNCTION set_order_item_price()
RETURNS TRIGGER
SECURITY DEFINER -- Executa com privilégios do dono da função
```

```
SET search_path = public
LANGUAGE plpgsql
AS $$
DECLARE
  v_price float8;
BEGIN
  -- Buscar o preço do produto
  SELECT "price" INTO v_price FROM public."products" WHERE "id" =
NEW."productId";

  IF v_price IS NULL THEN
    RAISE EXCEPTION 'Preço não encontrado para productId: %', NEW."productId";
  END IF;

  NEW."price" := v_price;
  RETURN NEW;
END;
$$;

DROP TRIGGER IF EXISTS before_insert_or_update_orderitem_price ON "OrderItem";

CREATE TRIGGER before_insert_or_update_orderitem_price
BEFORE INSERT OR UPDATE ON "OrderItem"
FOR EACH ROW
EXECUTE FUNCTION set_order_item_price();
```

3. Outra trigger para fazer a soma total automaticamente da order.
   Segue a trigger :

```
-- Função que calcula o total do pedido
CREATE OR REPLACE FUNCTION update_order_total()
RETURNS TRIGGER
SECURITY DEFINER
SET search_path = public
LANGUAGE plpgsql
AS $$
DECLARE
  v_total float8;
BEGIN
  -- Calcula o total somando (price * quantity) de todos os OrderItems do pedido
  SELECT COALESCE(SUM("price" * "quantity"), 0)
  INTO v_total
  FROM "OrderItem"
  WHERE "orderId" = COALESCE(NEW."orderId", OLD."orderId");

  -- Atualiza o campo total na tabela orders
  UPDATE "orders"
  SET "total" = v_total
  WHERE "id" = COALESCE(NEW."orderId", OLD."orderId");

  RETURN COALESCE(NEW, OLD);
END;
$$;

-- Trigger que dispara após INSERT, UPDATE ou DELETE em OrderItem
DROP TRIGGER IF EXISTS after_orderitem_change ON "OrderItem";

CREATE TRIGGER after_orderitem_change
AFTER INSERT OR UPDATE OR DELETE ON "OrderItem"
FOR EACH ROW
EXECUTE FUNCTION update_order_total();
```

Edge Functions:

Function para confirmação de conta enviando email:

```
// Setup type definitions for built-in Supabase Runtime APIs
import "jsr:@supabase/functions-js/edge-runtime.d.ts";
import { createClient } from 'jsr:@supabase/supabase-js@2';

const RESEND_API_KEY = Deno.env.get('RESEND_API_KEY');
const SUPABASE_URL = Deno.env.get('SUPABASE_URL');
const SUPABASE_SERVICE_ROLE_KEY =
Deno.env.get('SUPABASE_SERVICE_ROLE_KEY');

const handler = async (request: Request): Promise<Response> => {
  try {
    const { email } = await request.json();

    // Criar cliente Supabase com service_role para usar admin API
    const supabaseAdmin = createClient(
      SUPABASE_URL!,
      SUPABASE_SERVICE_ROLE_KEY!,
      {
        auth: {
          autoRefreshToken: false,
          persistSession: false
        }
      }
    );

    // Gerar link de confirmação de email
    const { data, error } = await supabaseAdmin.auth.admin.generateLink({
      type: 'signup',
      email: email,
    });

    if (error) {
      throw error;
    }

    // Obter o link de confirmação gerado
    const confirmationUrl = data.properties.action_link;

    // Enviar email via Resend com o link de confirmação
```

```
    const res = await fetch('https://api.resend.com/emails', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${RESEND_API_KEY}`,
      },
      body: JSON.stringify({
        from: 'no-reply@seudominio.com',
        to: email,
        subject: 'Confirmação de cadastro',
        html: `
          <div>
            <h2>Bem-vindo!</h2>
            <p>Obrigado por criar uma conta. Por favor, confirme seu email clicando no
botão abaixo:</p>
            <a href="${confirmationUrl}" style="display: inline-block; padding: 12px 24px;
background-color: #4CAF50; color: white; text-decoration: none; border-radius: 4px;">
              Confirmar Email
            </a>
            <p>Ou copie e cole este link no navegador:</p>
            <p>${confirmationUrl}</p>
          </div>
        `,
      }),
    });

    const emailData = await res.json();

    return new Response(JSON.stringify({ success: true, emailData }), {
      status: 200,
      headers: {
        'Content-Type': 'application/json',
      },
    });
  } catch (error) {
    return new Response(JSON.stringify({ error: error.message }), {
      status: 400,
      headers: {
        'Content-Type': 'application/json',
      },
```

```
    });
  }
};

Deno.serve(handler);
```

Para testar basta usar a rota de download no /docs

VIEWS

Uma view para mostrar a order completa com todos os detalhes de customers, products, orderItems e order.

```
CREATE VIEW order_details
WITH (security_invoker = true) AS
SELECT
  o."id" AS order_id,
  o."status",
  o."total",
  o."createdAt" AS order_date,
  o."customerId",
  c."name" AS customer_name,
  c."email" AS customer_email,
  oi."id" AS item_id,
  p."id" AS product_id,
  p."name" AS product_name,
  p."description" AS product_description,
  p."price" AS current_product_price,
  oi."quantity",
  oi."price" AS unit_price,
  (oi."quantity" * oi."price") AS item_total,
  oi."createdAt" AS item_created_at
FROM "orders" o
LEFT JOIN "customers" c ON o."customerId" = c."id"
LEFT JOIN "OrderItem" oi ON o."id" = oi."orderId"
LEFT JOIN "products" p ON oi."productId" = p."id";
```

Também pode ser testada pela rota de order details no /docs