

API DE ENTREGAS - ROCKETLOG

A API terá perfis de vendedor e cliente, com níveis de acesso definidos. Vamos implementar o envio de pedidos, gerenciar status como "processando", "enviado" e "entregue", e registrar as movimentações da entrega. Isso permitirá que os usuários acompanhem o status de suas entregas em tempo real.

Criando o projeto

1. package.json

- `npm init -y`
- 1.1 Configurando o package.json

2. express

Express é um framework (ou biblioteca) para Node.js usado para criar servidores e APIs web de forma simples, rápida e organizada.

- `npm i express`
- `npm i --save-dev @types/express`

3. pasta src

- 3.1 server.ts - servidor
- 3.2 app.ts - aplicação
 - 3.2.1 configurações básicas
 - Importa o framework Express, que facilita criar e gerenciar servidores HTTP no Node.js (como rotas, middlewares, etc)
 - Converte automaticamente o corpo das requisições (body) em JSON.
 - Exporta a aplicação app

4. typescript

- `npm i typescript @types/node -D`
- 4.1 tsx
 - `npm i tsx -D`
- 4.2 criando as configurações do tsconfig.json
 - `npx tsc --init`

- configuracoes padrao:

```
{
  "compilerOptions": {
    "module": "Node16",
    "target": "ES2022",
    "lib": ["ES2023"],

    "paths": {
      "@/*": [".src/*"]
    },

    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,

    "strict": true,
    "skipLibCheck": true,
  }
}
```

- 4.2.1
 - passando o "app" como "@/app" no server

5. middleware

Gerenciador de exeções na aplicação

- 5.1 -> criando a pasta middlewares

5.2 error-handling.ts

Para erros personalizados

- 5.3 AppError.ts
 - criando uma CLASSE para personalizar o erro e adicionamos a classe no middleware
- 5.4 express-async-errors (a aula diz para instalar)
 - OBS: Desde o v5 DO EXPRESS, ele trata promises rejeitadas/async handlers nativamente: você pode escrever async (req, res) e qualquer throw ou rejeição vai para o middleware de erro, ou seja: **NAO PRECISA INSTALAR**
- 5.5 passando o error-handling.ts para a aplicação

6. Zod

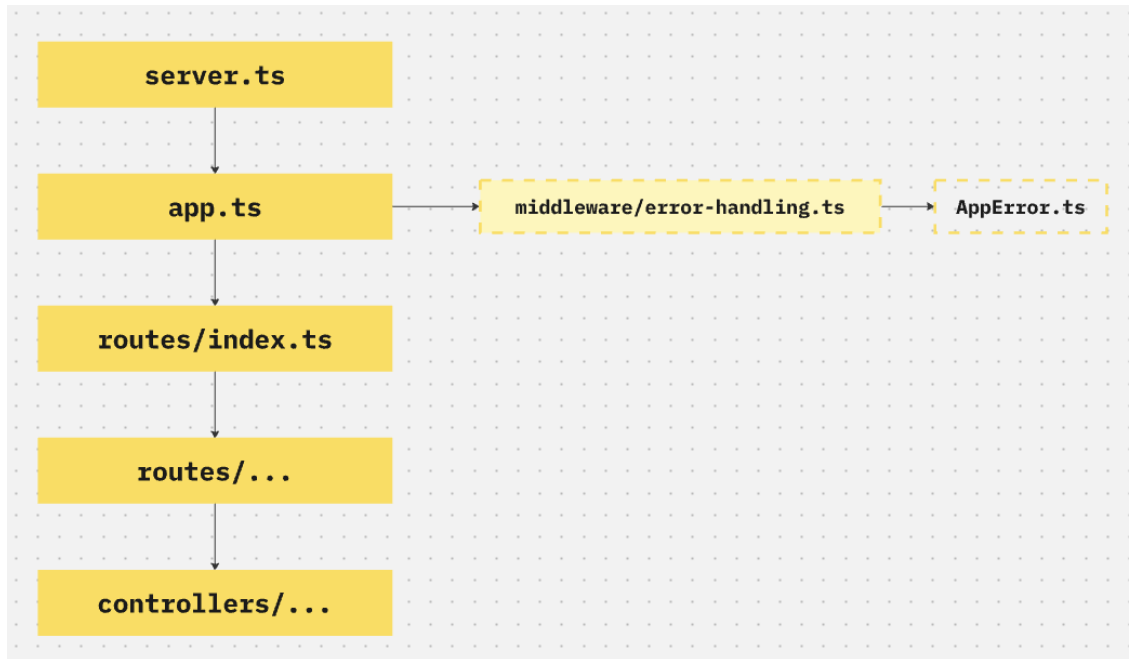
- O Zod serve para validar dados (ex: corpo de requisição, query params, variáveis de ambiente)
- E também para gerar tipos TypeScript automaticamente a partir dessas validações.
- `npm i zod`

7. Routes e Controller

ROUTES

Define os caminhos (rotas da API)

- É onde você diz “quando o cliente acessar tal URL, chame tal controller”.
 - 7.2 Router de Usuarios
 - 7.3 Router Index: centraliza todas as rotas aqui (users, products, orders...)
 - 7.3.1 Importamos os router para a aplicação: app.ts
 - HIERARQUIA DE ARQUIVOS:



CONTROLLERS

Recebe e responde requisições

- Os controllers são responsáveis por:
 - são responsáveis por lidar com as requisições e respostas HTTP
 - e devolver o resultado
 - **Define as CRUD**
 - 7.1 Controller para gerenciar Usuarios
 - 7.1.2 POST: metodo de criação de usuarios

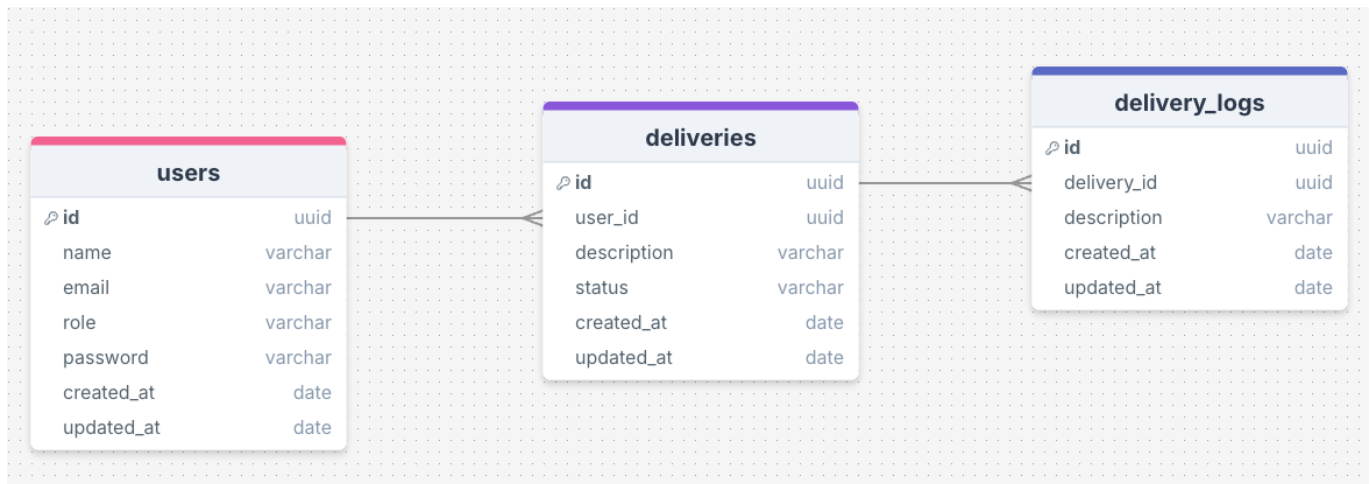
8. Criando projeto no Insomnia

- 8.1 New collection - Rocket Log
- 8.2 Base Enviroment -> Shared enviroment -> dev



- 8.3 Enviroment:
 - {"RESOURCE": "users"}

Banco de dados



- drawsql.app: <https://drawsql.app/teams/amanda-18/diagrams/api-de-entregas-rocketlog>

9. Docker

-> docker ps -a: lista todos os containers -> docker image ls: lista todas as imagens

- 9.1 Criação do Docker Compose
 - 9.1.1 Cria o arquivo na raiz do projeto e configura:

```

services:
  postgres:
    image: "bitnami/postgresql:latest"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: rocketlog
    ports:
      - "5432:5432"
  
```

- 9.1.2 Cria o docker compose `docker-compose up -d`
- 9.2 Rodando o banco de dados (beekeeper)

10. Prisma ORM

- 10.1 Instalando
 - `npm i prisma -D`
 - 10.1.1 Inicializando: inicializa o projeto com as configurações básicas para poder usar o Prisma com O postgres como banco de dados
 - `npx prisma init --datasource-provider postgresql`
 - muda a database_url =>


```
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/rocketlog?schema=public"
```
 - 10.1.2 adicionando no package.json a varivel de ambiente(env) no script
 - antes: `"dev": "tsx watch src/server.ts"`
 - depois: `"dev": "tsx watch --env-file .env src/server.ts"`

- 10.2 Modelo das tabelas (prisma/schema.prisma)

- 10.2.1 Usuarios

- Por padrao, todo novo usuario é um cliente (UserRole: client)

```
model User {
  id String @id @default(uuid())
  name String
  email String @unique
  password String

  role UserRole @default(client) // por padrao todo novo usuário é um client

  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime? @updatedAt @map("updated_at")

  deliveries Delivery[] // relacionamento com a tabela Delivery

  @@map("users")
}
```

- 10.2.2 Deliveries

- Por padrao, todo novo delivery esta "processando" (DeliveryStatus: processing)
 - A relação com a tabela de usuario é definida: `user User @relation(fields: [userId], references: [id])`
 - **OBS: A RELAÇÃO SEMPRE PRECISAR SER ESTABELECIDA NAS DUAS TABELAS QUE ESTÃO SE RELACIONANDO**
 - Adicionando a relação na tabela de User: `deliveries Delivery[]`

```
model Delivery {
  id String @id @default(uuid())
  userId String @map("user_id")
  description String

  status DeliveryStatus @default(processing) // por padrao toda nova entrega está com status processing

  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime? @updatedAt @map("updated_at")

  user User @relation(fields: [userId], references: [id]) // relacionamento com a tabela User
  logs DeliveryLog[] // relacionamento com a tabela DeliveryLog

  @@map("deliveries")
}
```

- 10.2.3 Delivery Log

- Relacionamento com o delivery: `delivery Delivery @relation(fields: [deliveryId], references: [id])`: cada delivery tem um delivery_log (status), mas um delivery log so pode estar associado a um delivery
 - **OBS: A RELAÇÃO SEMPRE PRECISAR SER ESTABELECIDA NAS DUAS TABELAS QUE ESTÃO SE RELACIONANDO**
 - Adicionando a relação na tabela de Delivery: `deliveryLogs DeliveryLog[]`

```
model DeliveryLog {
  id String @id @default(uuid())
  deliveryId String @map("delivery_id")
  description String

  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime? @updatedAt @map("updated_at")

  delivery Delivery @relation(fields: [deliveryId], references: [id]) // relacionamento com a tabela Delivery
  @map("delivery_logs")
}
```

- Roles e Status(enum): opções para uma das colunas da tabela

```
// ! Definindo as enums(opções) do banco de dados
// Definindo as roles de usuários(OPÇÕES DE USUARIOS)
enum UserRole {
  client
  sale
}

// Definindo os status de entrega(OPÇÕES DE STATUS)
enum DeliveryStatus {
  processing
  shipped
  delivered
}
```