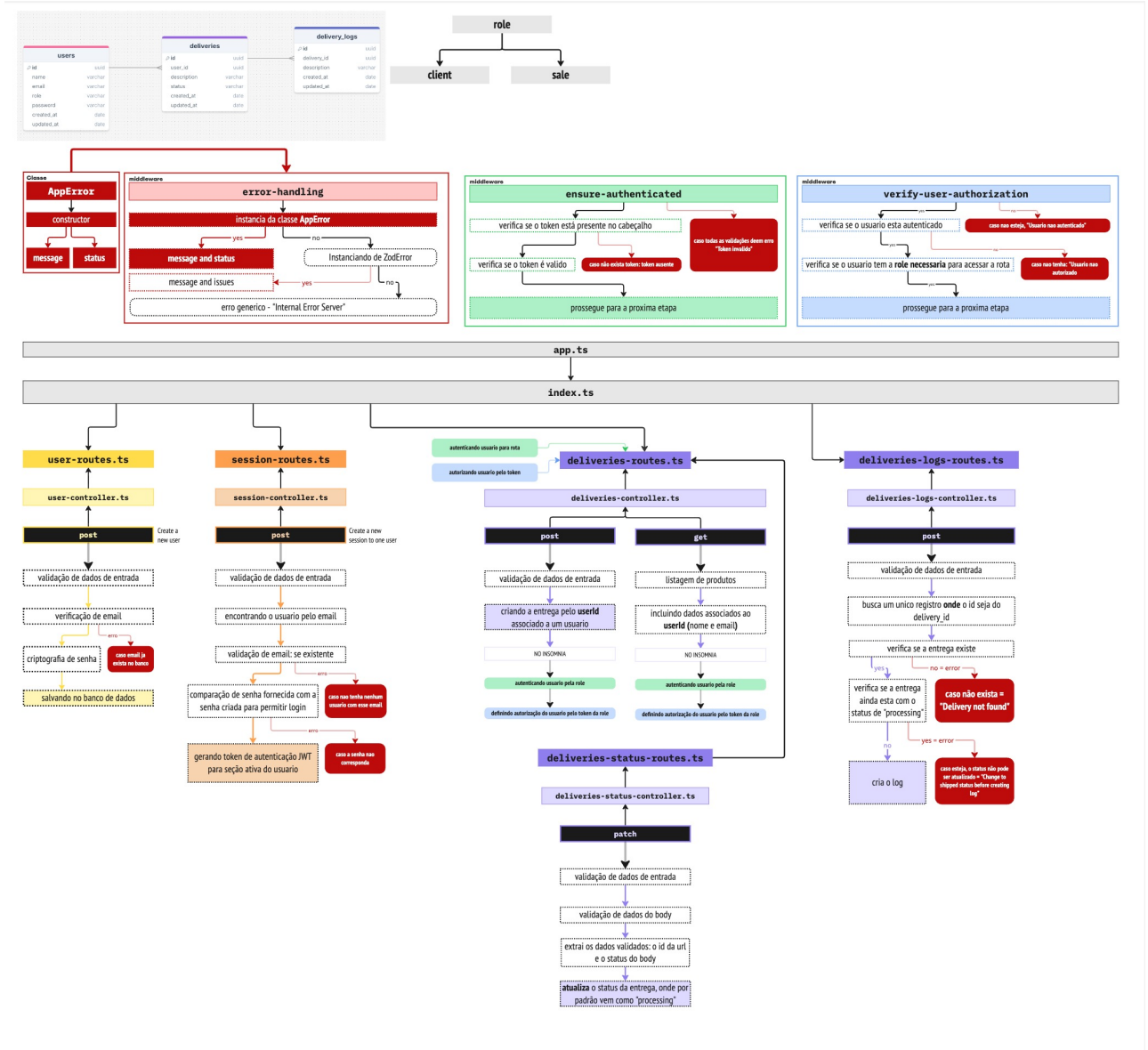


API DE ENTREGAS - ROCKETLOG

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 execuções 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

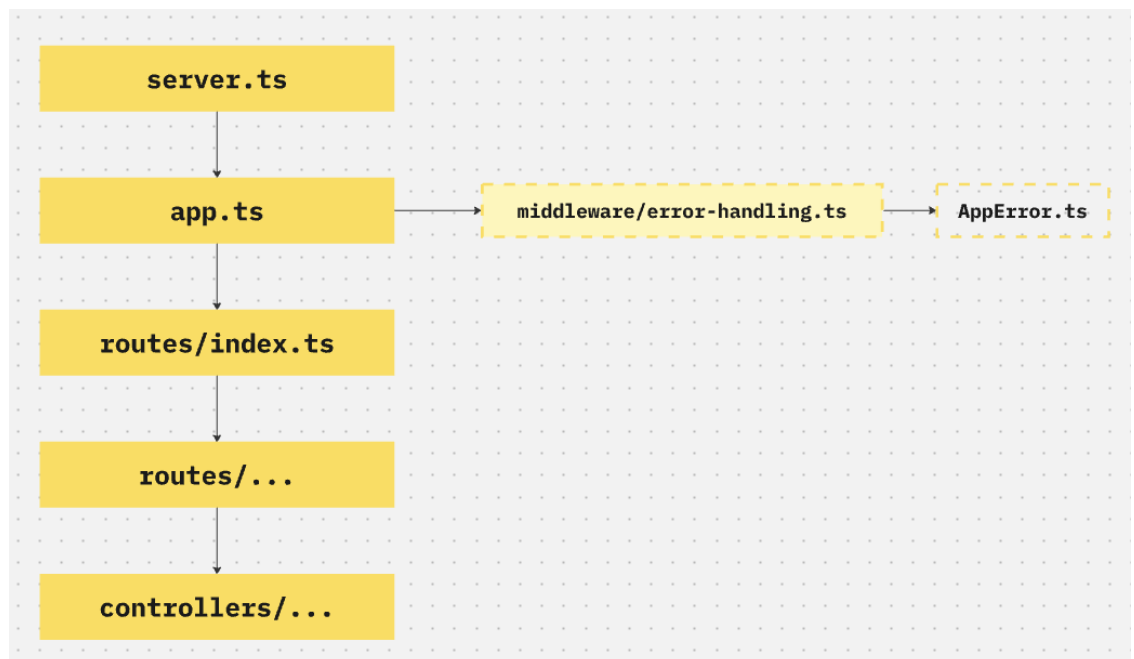
- O Zod serve para validar dados (ex: corpo de requisição, query params, variáveis de ambiente)
- E também para gerar tipos TypeScript automaticamente a partir dessas validações.
- `npm i zod`

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

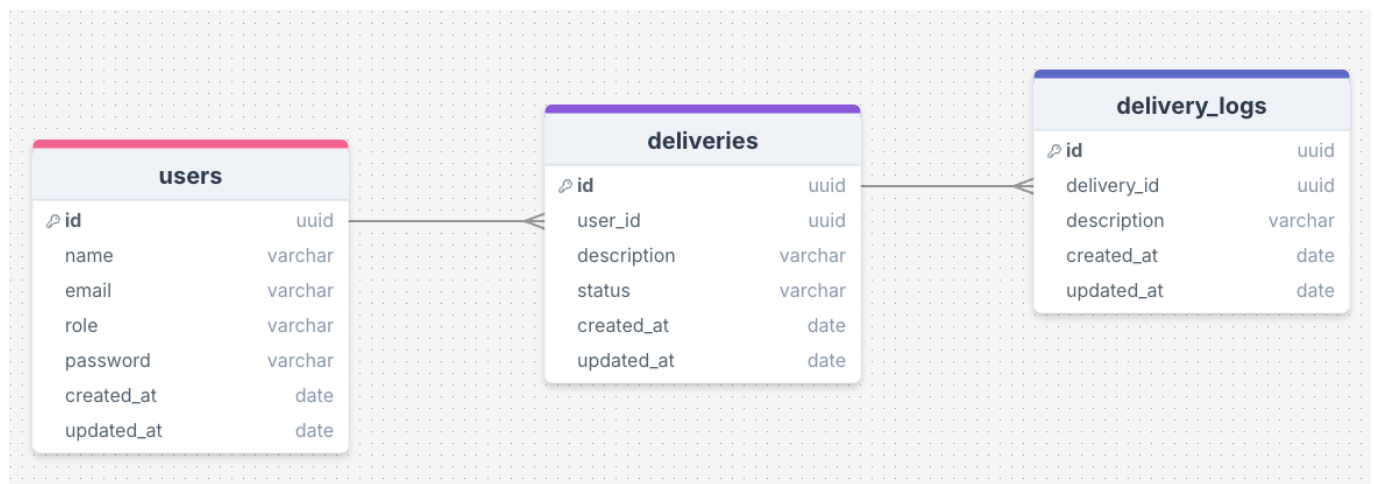
8. Criando projeto no Insomnia

- 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
}
```

- 10.3 Migrates
 - 10.3.1 criando as tabelas ja modeladas no prisma
 - `npx prisma migrate dev -> create-table` [nome do banco de dados]
 - visualizando as tabelas pelo Prisma Studio
 - 10.3.2 `npx prisma studio`

Usuários

11. ZOD - Validação

11.1 Validando dados do usuario

- 11.1.1 importando o zod para trabalhar com a validação de usuarios
- Definimos que os campos preenchidos:
 - `z.string()` = sejam string
 - `trim()` = ignorem espaços em branco
 - `min(x)` = tenham um minimo de characters
 - `email()` = seja do tipo email(zod tem essa propriedade)
- OBS => Quando alguém envia um formulário ou JSON (ex: { "name": "Juliete", "email": "juliete@email.com", "password": "1234" }), esses dados chegam dentro de `request.body`.

`const { name, email, password } = bodySchema.parse(request.body)` => 1. Pega o corpo da requisição (`request.body`); 2. Valida ele usando o schema (`bodySchema`); 3. Extrai (desestrutura) as variáveis `name`, `email` e `password` já validadas — prontas pra usar. 4. "Fazer o parse" é ler e interpretar um dado. `

```
create(request: Request, response: Response, next: NextFunction){
  // Validação dos dados de entrada usando Zod
  const bodySchema = z.object({
    name: z.string().trim().min(1),
    email: z.string().email(),
    password: z.string().min(6)
  });

  const { name, email, password } = bodySchema.parse(request.body);
```

11.2 Criptografando senha do usuario

- 11.2.1 BCrypt
 - instalando: `npm i bcryptjs`
 - instalando tipagem: `npm install -D @types/bcryptjs`
 - importamos e seguimos: `const hashedPassword = await hash(password, 8);`

12. Prisma Database

- configuração e inicialização do Prisma Client, que é o responsável por conectar sua aplicação Node.js ao banco de dados.
 - 12.1 importando o prisma client e configurando => database/prisma.ts
 - 12.2 importando o prisma database no controller de usuarios

13. Cadastrando usuarios no bando de dados

- 13.1 Verificando se o email ja existe, importando o AppError para personalizar a mensagem de erro

```
const userWithSameEmail = await prisma.user.findFirst({ where: { email } });
if (userWithSameEmail) {
  throw new AppError('Este email já está em uso.', 409);
}
```

- 13.2 Criando usuario no banco de dados

```
await prisma.user.create({
  data: {
    name,
    email,
    password: hashedPassword
  }
});
```

- 13.2.1 Nao retornando a senha no banco de dados
 - `const { password: _, ...userWithoutPassword } = user;`

- METHOD DE POST / CREATE

```
class UsersController {

  // 7.1.2 POST / CREATE
  async create(request: Request, response: Response, next: NextFunction) {

    // Validação dos dados de entrada usando Zod
    const bodySchema = z.object({
      name: z.string().trim().min(2),
      email: z.string().email(),
      password: z.string().min(6)
    });
    const { name, email, password } = bodySchema.parse(request.body);

    // 13.1 verificar se o email já existe no banco de dados
    const userWithSameEmail = await prisma.user.findFirst({ where: { email } });
    if (userWithSameEmail) {
      throw new AppError('Este email já está em uso.', 409);
    }

    // 11.2.2 Criptografar a senha antes de salvar no banco de dados
    const hashedPassword = await hash(password, 8);

    // 13.2 Lógica para salvar o usuário no banco de dados
    const user = await prisma.user.create({
      data: {
        name,
        email,
        password: hashedPassword
      }
    });

    // 13.2.1 Retornar o usuário criado (sem a senha)
    const { password: _, ...userWithoutPassword } = user;

    return response.json(userWithoutPassword);
  }
}
```

14. Criando controller e rota de autenticação

- criando o arquivo `session-controller.ts` + `session-routes.ts` => **importamos o routes no index.ts**
- criamos no insomnia a pasta sessions e o environment para "RESOURCE": "sessions"
- 14.1 VALIDAÇÕES
 - 14.1.2 Usamos o email para encontrar o usuario
 - `const user = await prisma.user.findFirst({ where: { email, password } });`
 - 14.1.3 Caso 1: nao tem nenhum usuario com esse email
 - `if (!user) { throw new AppError("Email ou senha inválidos", 401);}`

- 14.1.4 caso 2: Comparação da senha fornecida com a senha armazenada no banco de dados. Em um cenário real, a senha armazenada deve ser hashada e você deve usar uma função de comparação de hash

- `const passwordMatches = await compare(password, user.password);`
- 14.1.4.1 Caso a senha não corresponda
 - `if (!passwordMatches) {throw new AppError("Email ou senha inválidos", 401);}`

15. TOKEN DE AUTENTICAÇÃO

- 🗝️ O que é o “token do usuário”
- Depois que o usuário faz login (ou seja, envia e-mail e senha corretos), o servidor precisa de uma forma segura de reconhecer esse usuário nas próximas requisições — sem que ele precise enviar a senha toda hora.
- 👉 Aí entra o token, que é um código digital único (geralmente um JWT, JSON Web Token). Ele representa a sessão autenticada desse usuário.

- 15.1 JSON WEB TOKEN

- instalando: `npm i jsonwebtoken`
- 15.1.2 criando pasta **configs** -> **auth.ts**

```
export const authConfig = {  
  jwt: {  
    secret: process.env.JWT_SECRET,  
    expiresIn: '1d',  
  },  
}
```

- Esse trecho de código define a configuração de autenticação com JWT (JSON Web Token) — uma forma muito comum de autenticar usuários em aplicações Node.js.
- `expiresIn: '1d'`
 - Indica o tempo de expiração do token.
 - '1d' = 1 dia (você também pode usar '2h', '30m', etc.).
 - Depois desse tempo, o usuário precisa fazer login de novo para gerar outro token.
- 15.1.3 .env => JWT_SECRET= chave_super_secreta
 - ** tem um site chamado hash-generate que gera hashes de segurança

- 15.2 passando o token de autenticação para o controller

- 15.2.1 sign() => A função sign() do pacote jsonwebtoken serve para gerar um token JWT — ou seja, criar o token que **identifica um usuário** autenticado.
 - precisamos instalar a tipagem = `npm i --save-dev @types/jsonwebtoken`
- 15.2.1 o secret nao pode ser undefined, temos que resolver:

```
const secret: string | undefined  
secret, {
```

- 15.2.1.a
- 15.2.1.b criamos o arquivo env.ts e passamos as configurações
- 15.2.1.c no auth.ts importamos e mudamos as configurações do secret

```
import { env } from "../env";  
  
export const authConfig = {  
  jwt: {  
    secret: env.JWT_SECRET,  
    expiresIn: '1d',  
  }  
}
```

- 15.3 retornando a resposta completa da sessão com o usuário autenticado

Autenticação e Autorização

16. Autenticação

16.1 Middleware de autenticação de token

- Vai verificar se o usuário está autenticado antes de permitir o acesso a certas rotas.

```
// 16.1 Middleware de autenticação de token

// importações
import { Request, Response, NextFunction } from 'express';
import { verify } from 'jsonwebtoken';

import { authConfig } from '@/configs/auth';
import { AppError } from '@/utils/AppError';

// 16.1.2 Interface para o formato do payload do token
interface TokenPayload {
  role: string;
  sub: string;
}

// 16.1.3 Middleware para garantir que o usuário está autenticado
function ensureAuthenticated(request: Request, response: Response, next: NextFunction) {
  try {
    // a. Verificar se o token está presente
    const authHeader = request.headers.authorization;
    if (!authHeader) {
      throw new AppError('Token ausente.', 401);
    }

    // b. Verificar se o token é válido
    const [, token] = authHeader.split(' '); // Bearer <token>

    const { role, sub: user_id } = verify(token, authConfig.jwt.secret) as TokenPayload;
    request.user = {
      id: user_id,
      role
    };

    // c. Prosseguir para o próximo middleware ou rota
    return next();
  } catch (error) {
    throw new AppError('Token inválido.', 401);
  }
}
```

- 16.1.1 Importações
 - verify → função do jsonwebtoken, usada pra validar o token JWT enviado pelo usuário.
 - authConfig → normalmente contém a chave secreta (secret) e o tempo de expiração do token.
 - AppError → uma classe customizada pra lançar erros padronizados (ex: mensagens e status HTTP).
- 16.1.2 Interface
 - Essa interface define o formato dos dados dentro do token JWT (as "claims"). Por exemplo: role → papel do usuário (ex: client, sale, admin etc.), sub → normalmente o ID do usuário (vem do "subject" do token JWT).

**** payload:** É o conteúdo que você define ao gerar o token, com informações sobre o usuário.


```
{
  "sub": "1234",           // ID do usuário (subject)
  "role": "client",       // papel do usuário
  "name": "Amanda",
  "iat": 1706486082       // data em que o token foi criado (issued at)
}
```

- 16.1.3 Middleware
- Funções:
 - Ler o token JWT do cabeçalho da requisição (geralmente Authorization: Bearer).
 - Verificar se o token é válido usando verify() do jsonwebtoken.
 - Decodificar o token e extrair o sub (ID do usuário) e o role.
 - Anexar essas informações ao request, pra que os controllers saibam quem é o usuário logado.
 - Chamar next() se estiver tudo certo, ou lançar AppError se o token estiver ausente ou inválido.
- FORMATO DO TOKEN NO CABEÇALHO: **Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...**
 - a. verifica se o token esta presente no cabeçalho
 - b. se estiver presente, verifica se é valido
 - c. seguir para a proxima etapa

- EM RESUMO

◆ 1. Separando o token


ts

 Copiar código

```
const [, token] = authHeader.split(' ');
```

O header vem assim:


makefile

 Copiar código

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...
```

Ao fazer `split(' ')`, ele gera um array:

js

 Copiar código


```
['Bearer', 'eyJhbGciOiJIUzI1NiIsInR5cCI6...']
```

E o código usa **desestruturação** para pegar só a segunda parte, que é o token.

Então `token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6...'`.

◆ 2. Verificando e decodificando o token

ts

 Copiar código

```
const { role, sub: user_id } = verify(token, authConfig.jwt.secret) as TokenPayload;
```

Aqui o `verify()`:

- **valida** se o token foi assinado com a **mesma chave secreta** (`authConfig.jwt.secret`);
- **decodifica** o conteúdo (o *payload*);
- **lança erro automaticamente** se o token for inválido, expirado ou adulterado.

O resultado contém os dados que estavam dentro do token, por exemplo:

json


 Copiar código

```
{
  "sub": "c43a7d41-f87b-4e8a-8f9c-1e0d7df4e345",
  "role": "client",
  "iat": 1730000000
}
```

Depois ele renomeia `sub` → `user_id` e extrai `role`.

◆ 3. Salvando o usuário dentro da requisição

ts

 Copiar código

```
request.user = {  
  id: user_id,  
  role  
};
```

Aqui o middleware anexa as informações do usuário autenticado ao objeto `request`, criando `request.user`.

Assim, qualquer rota ou controller pode acessar:

ts

 Copiar código

```
request.user.id // ID do usuário logado  
request.user.role // papel do usuário
```

Isso evita precisar decodificar o token várias vezes.

🧠 Em resumo:

Essas linhas:

1. Extraem o token JWT do cabeçalho.
2. Validam e decodificam o token.
3. Guardam o ID e o papel do usuário na requisição.

Depois disso, o middleware pode chamar:

ts

 Copiar código

```
return next();
```



pra permitir o acesso à rota.

** no request user... ele vai reclamar de tipagem, nesse momento precisamos criar uma tipagem para ele

- o 16.1.3.1 types/express.d.ts

```
declare namespace Express {  
  export interface Request {  
    user?: {  
      id: string;  
      role: string;  
    }  
  }  
}
```

- o

17. Deliveries

- 17.1 Criando controller e o arquivo base com metodo post

No insomnia:

- 1. Criamos a pasta de Deliveries
- 2. Criamos o metodo post
- 3. definimos o enviroment - sempre
- 4. em "auth" habilitamos a utilização do token e fazemos a requisição pegar automaticamente o valor correspondente para retornar

Edit Tag

Function to Perform
Response – reference values from other request's responses

Attribute
Body Attribute – value of response body

Request
[Sessions] POST Create

Filter (JSONPath or XPath)
\$.token

Trigger Behavior
Never – never resend request

Live Preview
refresh
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyb2xlIjoieY2xpZW50IiwiaWF0IjoxNzYxNDE2ODkwLCJleHAiOjE3NDMyOTAsInYiOiI6IjYzYzYwLTA5OTYtNDZmNy05NzFkLTQ4ZjRiM2QwYTcwYyJ9.0oONiOK2QJ4YNhmah-d9Fd5A0OB2b6Mc98-ttAd6XxM

Done

- o token precisa vir do metodo post onde ele foi criado

- 17.2 Criando a rota

- 17.2.1 na rota adicionamos o `ensureAuthenticated` para usuarios autenticados

18. Autorização

Vamos verificar se o usuario tem autorização para consultar oq ele esta tentando consultar

- 18.1.1 `verify-user-authorization.ts`
- 18.1.2 dentro de `deliveries-routes.ts` passamos o middleware de autorização
`deliveriesRoutes.post("/", ensureAuthenticated, verifyUserAuthorization(['sale']), deliveriesController.create);`
 - Dentro dele passamos o array de que tipo de role esta autorizada a entrar em URL com rotas `/deliveries`

Delivery

18. Cadastrando entregas

- 18.1 Metodo POST - CRIANDO
 - 18.1.1 Validação de dados de entrada como sempre. Somente para reforçar, a entrada de dados é o formato e quantidade de caracteres que o campo vai receber.
 - 18.1.2 Criando o "delivery" no banco de dados. Logica para criar o delivery no banco de dados. Cada delivery vai estar associado a um usuario, pelo userId. O userId e o description sao colunas definidas no momento da criação da tabela **Delivery**.
 - a. Nesse momento de teste no insomnia, estamos com uma sessao de vendedor aberta, entramos no metodo post do delivery e abrimos manualmente um delivery no userId da usuaria Juliete
- 18.2 Metodo GET - LISTANDO
 - 18.2.1 `const deliveries = await prisma.delivery.findMany();`
 - 18.2.2 criando a rota de get no deliveries-routes.ts
 - a. no insomnia criamos o metodo get, pasamos o base_enviroment e o resource como URL, em seguida passamos no auth bearer o token de autorização e ai sim liberamos o metodo para listar os deliveries para o usuario da sessão de sale
 - 18.2.3 incluindo alguns dados do usuario associado a entrega na listagem dos produtos

```
const deliveries = await prisma.delivery.findMany({
  include: {
    user: { select: { name: true, email: true } },
  },
});
```

▪ a.

19. Status do pedido

- 19.1 validação de dados de entrada e validação de body
- 19.2 Logica para atualizar o status da entrega no banco de dados

```
await prisma.delivery.update({
  data: {
    status: status, // "é o status que quero atualizar"
  }, where: {
    id: id, // "é o id da entrega que quero atualizar"
  },
});
```

- 19.2.1 no insomnia criamos o metodo de patch - atualizar
- 19.3 passando o controller para o routes de deliveries com o :id do delivery como parametro da url
 - `deliveriesRoutes.patch("/:id/status", deliveriesStatusController.update);`

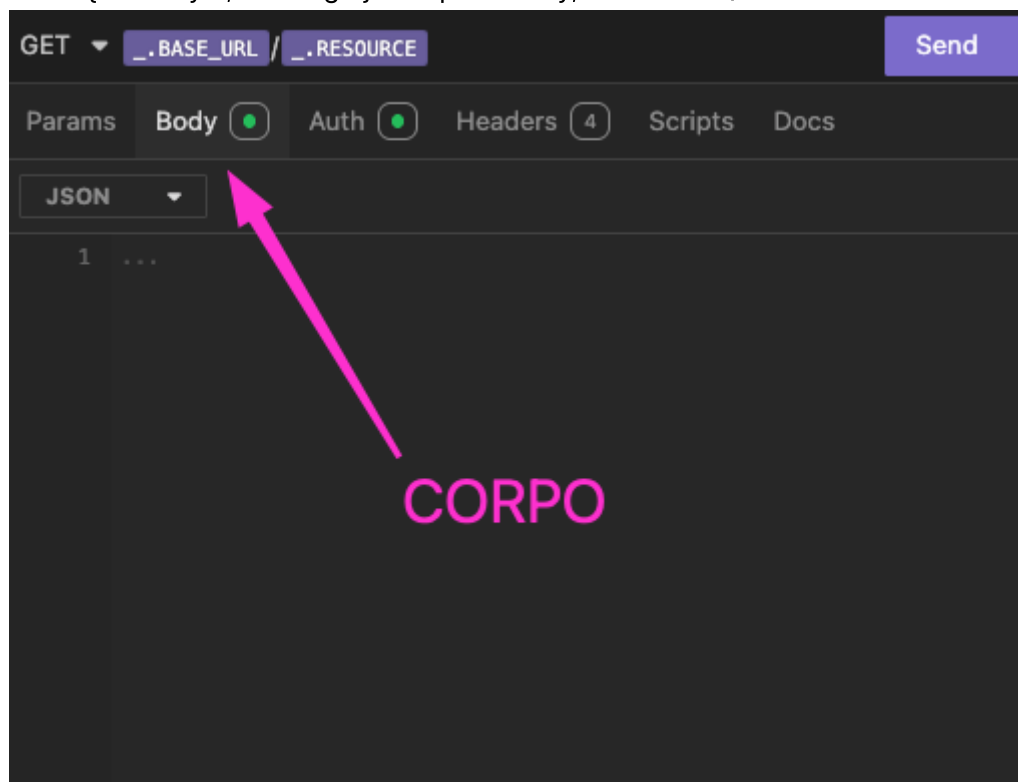
20. Log de Entregas

Logs de entrega, permitindo adicionar informações sobre o status de envios

- 20.1 CREATING

- **UMA COISA BEM CLARA QUE EU SO ENTENDI AGORA:**

- `const { deliveryId, message } = request.body; => ISSO AQUI VEM DO CORPO:`



- 20.1.1 passando a rota para o delivery-logs-routes.ts

- queremos dar acesso apenas ao vendedor para CRIAR a mensagem de log
 - mas queremos dar acesso ao cliente para VISUALIZAR a mensagem de log

- 20.1.2 passando o delivery-logs-routes.ts para o index.ts

- 20.1.3 criando a rota no insomnia

- 20.1.4 CRIANDO A LOGICA

- a. validação de dados de entrada
 - b. `findUnique()` => buscar um único registro no banco de dados com base em um campo que seja único — normalmente o id, email ou qualquer outro campo marcado com `@unique` no schema.
 - c. verifica se a entrega existe
 - d. verifica se o status é processing, se for, não tem como ter logs de envios porque ainda está processando, os logs veem a partir do shipped
 - e. aí cria o log

- 20.2 SHOWING

- mostrando os logs de envio para o usuário
 - a. validação de dados

- b. busca um unico registro ONDE o id seja igual ao delivery_id
- c. Verifica se o usuário é um cliente e se ele é o proprietário da entrega
- d. mostra todos os logs da entrega para o usuario
- 20.3 CONDIÇÃO: se um pedido ja foi entregue, ele não pode receber novos logs
- 20.4 Registrando no Log Alteração do Status

JEST - Testes Automatizados

- 21. instalação e configuração
 - jest: `npm i jest @types/jest ts-jest -D`
 - supertest: `npm i supertest@7.0.0 @types/supertest@6.0.2 -D`
 - ts-node: `npm i ts-node -D`
 - 21.1 Configurando jest
 - `npx jest --init`
 - respondemos algumas perguntas de configuração
 - configuramos o arquivo que foi criado: **jest.config.ts**

```
import type {Config} from 'jest';

const config: Config = {
  bail: true,
  clearMocks: true,
  coverageProvider: "v8",
  preset: "ts-jest",
  testEnvironment: "node",
  testMatch: ["<rootDir>/src/**/*.test.ts"],
  moduleNameMapper: {
    "^@/(.*)$": "<rootDir>/src/$1"
  },
};

export default config;
```

- package.json: criando o script => `"test": "NODE_OPTIONS='--experimental-vm-modules' jest --watchAll --runInBand"`
- 22. TESTES => src/tests
- TEST: Rodar testes e validar lógica do código
- SUPertest: Com Supertest você testa rotas/requests/respostas da API. Ele simula requisições HTTP sem precisar de Postman ou servidor externo.
 - 22.1 criando o supertest para o user-controller: **user-controller.test.ts**

- a. teste de criação de usuario: ele vai criar um novo usuario de exemplo para poder fazer o teste
- `const response = await request(app).post("/users").send...`



Quebra da linha

- `request(app)`
Diz: "quero fazer uma requisição para a aplicação Express"
(sem subir servidor de verdade — Supertest faz isso internamente)
- `.post("/users")`
Diz: "faz uma requisição POST para a rota `/users`"
- `.send({ ... })`
Diz: "envia esse corpo JSON na requisição"
Igual quando você manda um body no Insomnia:

- 22.1.1 limpando o usuario de exemplo depois de ter executado todos os testes

TESTES

- 22.1.2 teste de erro para emails duplicados
- 22.1.3 teste para email digitado de forma errada
- 22.2 criando o supertest para a autenticação da sessao: session-controller
 - 22.2.1 ANTES de todos os testes: cria um usuário para testar a sessão
 - 22.2.2 DEPOIS de todos os testes, deletamos o usuario

TESTES

- 22.2.3 Teste de criação de sessao
- OS TESTES SAO BEM REPETITIVOS, ENTÃO NAO APLICAMOS TESTES PARA TUDO NESSE PROJETO

AO LONGO DO DESENVOLVIMENTO

PROBLEMA AO INICIAR PROJETO

CAUSA 1: DOCKER DESCONECTADO

Desligou o computador e o docker fechou sozinha

1. abra o aplicativo do docker
2. rode `docker compose up -d`
3. depois rode todos os outros scripts: a. `npm run dev` b. `npx prisma studio` c. `npm run test`