UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Boas práticas em API REST

QXD0279 - Desenvolvimento de Software para Web 2

Prof. Bruno Góis Mateus (brunomateus@ufc.br)

Agenda

- Introdução
- Estruturando o projeto da API
- Padronizando respostas
- Validação de dados
- Documentando uma API

Introdução

Introdução

🧠 Por que estudar boas práticas em APIs?

- Criar uma API funcional é fácil.
- Criar uma API sólida, escalável e fácil de manter é outra história

Introdução

🚀 O Express é extremamente flexível

- Ao contrário de frameworks mais “opinionados” como NestJS, AdonisJS ou Laravel, o Express não impõe estrutura.
- Isso significa:

✅ Liberdade total

✅ Baixa curva de aprendizado

❌ Maior risco de bagunça

❌ Código inconsistente entre times

Introdução

Diversas formas de estruturar um projeto Express

- Estilo MVC
  - controllers/
  - services/
  - repositories/
- Estilo por domínios/módulos
  - modules/users/
  - modules/products/
- Arquiteturas avançadas
  - Clean Architecture
  - Hexagonal Architecture
  - DDD (Domain-Driven Design)

Introdução

Express não define

- ❌ como criar camadas
- ❌ como tratar erros
- ❌ como validar inputs
- ❌ como paginar
- ❌ como versionar
- ❌ como documentar endpoints
- ❌ como estruturar controllers
- ❌ como criar respostas padronizadas
- ❌ como lidar com status codes
- ❌ como lidar com repositórios/banco
- ❌ como lidar com async/await e erros

Introdução

🌎 O que é uma API REST bem construída?

1️⃣ Intuitiva: Rotas previsíveis e consistentes

2️⃣ Segura: Validação, sanitização e tratamento de erros

3️⃣ Estável: Contratos claros que não quebram a cada mudança

4️⃣ Documentada: Swagger / OpenAPI

5️⃣ Observável: Códigos HTTP corretos e estruturados

6️⃣ Padronizada: Respostas com o mesmo formato em toda a API

Estruturando o projeto da API

Estruturando o projeto da API

Por que a estrutura é tão importante?

- Uma API mal estruturada:
  - ❌ se torna difícil de manter
  - ❌ acumula código duplicado
  - ❌ torna testes mais difíceis
  - ❌ dificulta escalar o sistema
  - ❌ força novos devs a reaprender padrões
- Uma API bem estruturada:
  - ✅ facilita manutenção
  - ✅ separa responsabilidades
  - ✅ melhora testabilidade
  - ✅ permite evolução sem bagunça
  - ✅ organiza a lógica de negócio

Estruturando o projeto da API

Estrutura mínima

- Usado apenas para APIs pequenas ou protótipos.

```
src/
  ├── app.ts
  └── index.ts
```

Estruturando o projeto da API

Estrutura por rotas

- Boa para APIs pequenas, mas não separa lógica de negócio

```
src/
  ├── routes/
  ├── middlewares/
  └── app.ts
```

Estruturando o projeto da API

Estrutura por funcionalidade / domínio

- Cada módulo contém tudo (rota, controller, serviço…)
- Muito usada em projetos grandes

```
src/
  ├── users/
  ├── products/
  ├── orders/
```

Estruturando o projeto da API

A estrutura que usaremos: MVC (Estendido)

- Em uma API REST não possuímos a camada de View, então usamos o MVC adaptado:
- Controller
  - Recebe a requisição → chama o serviço → retorna a resposta
- Service
  - Lógica de negócio da aplicação
- Repository
  - Comunicação com o banco (ORM)
- Model/Entity
  - Representação da tabela (TypeORM)

Estruturando o projeto da API

A estrutura que usaremos: MVC (Estendido)

```
src/
├── config/              ← config geral (db, env, swagger…)
├── modules/
│   ├── users/
│   │   ├── user.controller.ts
│   │   ├── user.service.ts
│   │   ├── user.repository.ts
│   │   ├── user.entity.ts
│   │   ├── user.routes.ts
│   │   └── user.schema.ts (Zod)
│   └── products/
├── middlewares/
├── shared/
│   ├── errors/
│   └── responses/
├── app.ts
└── server.ts
```

Estruturando o projeto da API

A estrutura que usaremos: MVC (Estendido)

- Controller
  - Não contém lógica de negócio
  - Lê inputs
  - Chama o service
  - Retorna resposta padronizada

Estruturando o projeto da API

A estrutura que usaremos: MVC (Estendido)

- Service
  - Contém a lógica de negócio
  - Orquestra validações
  - Pode falar com vários repositórios
  - Onde regras são implementadas

Estruturando o projeto da API

A estrutura que usaremos: MVC (Estendido)

- Repository
  - Comunicação com o banco via TypeORM
  - find, save, delete, queries personalizadas

Estruturando o projeto da API

A estrutura que usaremos: MVC (Estendido)

```
//Fluxo da requisição
Request → Middleware → Controller → Service → Repository → Banco

//Fluxo da resposta
Repository → Service → Controller → Resposta padronizada → Cliente
```

Padronizando respostas

Padronizando respostas

O problema: respostas inconsistentes

- APIs tornam-se difíceis de consumir
  - Clientes precisam tratar casos especiais o tempo todo
- Aumenta o acoplamento entre frontend e backend
  - O frontend precisa “adivinhar” o formato da resposta a cada endpoint
- Dificulta a documentação e gera dúvidas
- Complica o tratamento global de erros

Padronizando respostas

Exemplos de API inconsistente

```
// endpoint A
{ "user": { "name": "Ana" } }

// endpoint B
{ "data": { "product": "Notebook" } }

// endpoint C
{ "message": "ok", "name": "Pedro" }

// endpoint D (erro)
{ "status": 400, "msg": "invalid input" }
```

Padronizando respostas

O objetivo do formato consistente

✅ Tornar previsível

✅ Tornar fácil de consumir

✅ Padronizar sucesso e erro

✅ Simplificar testes

✅ Simplificar documentação (Swagger)

Padronizando respostas

Exemplos de padrões usados em APIs

```
// Padrão minimalista
{
  "data": { ... }
}
// Padrão minimalista com metadados
{
  "success": true,
  "data": { ... }
}
//Padrão "do Google"
{
  "data": { ... },
  "error": null
}
```

Padronizando respostas

Exemplos de padrões usados em APIs

```
//Padrão enriquecido
{
  "success": true,
  "data": { ... },
  "errors": [],
  "meta": {
    "timestamp": "...",
    "path": "/api/v1/users"
  }
}
```

Padronizando respostas

```
//Resposta de sucesso
{
  "success": true,
  "data": {},
  "meta": {}
}
//Resposta de erro
{
  "success": false,
  "error": {
    "message": "Validation failed",
    "details": [],
    "code": "BAD_REQUEST"
  },
  "meta": {}
}
```

Validação de dados

Validação de dados

Por que validação é importante?

- Evita falhas lógicas no backend
  - Ex.: operações com campos faltando ou formatos inválidos
- Protege sua aplicação
  - Entrada inválida é vetor comum de ataques (ex.: injections, payloads malformados)
- Evita erros difíceis de rastrear
  - Validação clara → erros previsíveis → menor custo de manutenção
- Melhora a experiencia do usuário
  - Mensagens de erro claras ajudam frontend a corrigir inputs antes mesmo de enviar

Validação de dados

O que acontece quando NÃO validamos os dados?

- Exemplos de problemas reais:
  - Operações com valores undefined causam erros inesperados
  - Scripts maliciosos podem chegar ao banco de dados
  - O backend fica cheio de ifs e checagens manuais
  - Controllers se tornam enormes e pouco legíveis
  - A API retorna erros inconsistentes e difíceis de testar

Validação de dados

Validar manualmente é difícil e repetitivo

- Muito código repetido
- Não há reaproveitamento
- Fica difícil atualizar o schema
- Fácil de esquecer validações
- Erros inconsistentes

```
if (!req.body.email) { ... }
if (!req.body.email.includes("@")) { ... }

if (typeof req.body.age !== "number") { ... }
if (req.body.age < 18) { ... }

if (!Array.isArray(req.body.tags)) { ... }
```

Validação de dados

Bibliotecas mais usadas no mercado

🔷 Zod

- Tipos inferidos automaticamente para TypeScript
- Sintaxe clara
- Ótimo para APIs modernas

🔷 Yup

- Popular no frontend, especialmente com React
- Muita compatibilidade com forms

🔷 class-validator

- Muito usado junto com TypeORM
- Baseado em decorators
- Bom para aplicações OOP

Validação de dados

Exemplo básico com Zod

```
import { z } from "zod";

const userSchema = z.object({
  name: z.string().min(3),
  email: z.string().email(),
  age: z.number().int().min(18)
});
```

Validação de dados

Exemplo básico com Zod

```
import { z } from "zod";

const userSchema = z.object({
  name: z.string().min(3),
  email: z.string().email(),
  age: z.number().int().min(18)
});

//Validação
userSchema.parse(req.body);
```

Validação de dados

Exemplo de mensagens de erro do Zod

```
//Entrada
{
  "name": "A",
  "email": "invalido",
  "age": 15
}
//Resultado da validação
[
  { path: ["name"], message: "String must contain at least 3
character(s)" },
  { path: ["email"], message: "Invalid email" },
  { path: ["age"], message: "Number must be greater than or equal to
18" }
]
```

Validação de dados

Criando um middleware de validação

```
import { z } from "zod";
export const validate = (schema: z.Schema) => (req, res, next) => {
    try {
      schema.parse(req.body);
      next();
    } catch (err) {
      return res.status(400).json({
        success: false,
        error: {
          message: "Validation failed",
          details: err.errors,
          code: 400
        },
        meta: {
          path: req.path,
          timestamp: new Date().toISOString()
        }
      });
    }
```

Validação de dados

Usando o middleware em uma rota

```
import { Router } from "express";
import { z } from "zod";
import { validate } from "../middlewares/validate";

const router = Router();
const createUserSchema = z.object({
  name: z.string().min(3),
  email: z.string().email(),
  password: z.string().min(6)
});
router.post(
  "/users",
  validate(createUserSchema),
  userController.create
);
```

Validação de dados

Validando Params e Query

```
const paramsSchema = z.object({
  id: z.string().uuid(),
});

const querySchema = z.object({
  page: z.string().optional(),
  limit: z.string().optional()
});

router.get(
  "/users/:id",
  validate(paramsSchema),
  validate(querySchema),
  userController.getUser
);
```

Validação de dados

Tipando

- Agora CreateUserDTO é totalmente seguro, porque:
  - É IMPOSSÍVEL divergir do schema
  - Se mudar a validação, o tipo muda junto

```
export type CreateUserDTO = z.infer<typeof createUserSchema>;
```

Validação de dados

Atualizando o middleware

```
import { z } from "zod";
export const validate = (schema: z.Schema) => (req, res, next) => {
    try {
      req.validated = schema.parse(req.body);
      next();
    } catch (err) {
      return res.status(400).json({
        success: false,
        error: {
          message: "Validation failed",
          details: err.errors,
          code: 400
        },
        meta: {
          path: req.path,
          timestamp: new Date().toISOString()
        }
      });
    }
```

Validação de dados

Tipando

```
// types/index.d.ts
import * as express from 'express';

declare global {
  namespace Express {
    interface Request {
      validated?: any;
    }
  }
}
```

Documentando uma API

Documentando uma API

Por que documentar uma API?

✔ Facilita a comunicação entre equipes

- Frontend, backend, QA e DevOps precisam de um contrato claro

✔ Evita suposições e erros de integração

- Se a documentação é fraca, cada time interpreta a API de um jeito

✔ Ajuda novos desenvolvedores a entender o sistema

- Onboarding mais rápido, menos dúvidas, menos gargalos

Documentando uma API

Por que documentar uma API?

✔ Permite testes automatizados e ferramentas de inspeção

- Ex.: Postman, Insomnia, Swagger UI.

✔ Garante consistência durante a evolução da API

- Evita endpoints “escondidos” e comportamentos inesperados.

Documentando uma API

Problema com documentação manual

- Documentação manual significa:
  - Arquivos .md que ficam desatualizados
  - Escrever payloads e erros manualmente
  - Não garantir consistência entre documentação e código
  - Duplicação de trabalho
  - Difícil acompanhar mudanças

➡ Precisamos de documentação sincronizada com o código.

Documentando uma API

OpenAPI + Swagger: por que usar?

✔ Padrão da indústria

- OpenAPI é o formato universal para documentar APIs REST.

✔ Compatível com várias ferramentas

- Swagger UI, Postman, Insomnia
- GitHub API renderer
- Geradores de SDK

✔ Permite documentação viva

- Atualiza automaticamente conforme a API cresce.

Documentando uma API

Usando a zod-to-openapi

```
npm install swagger-ui-express @asteasolutions/zod-to-openapi

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

Documentando uma API

Usando a zod-to-openapi - Schema

```
import { z } from 'zod'
import { extendZodWithOpenApi } from '@asteasolutions/zod-to-openapi';

extendZodWithOpenApi(z)

export const createProduct = z.object({
  name: z.string('O campo nome é obrigatório').openapi({ example: "Sanduicheira Elétrica"}),
  description: z.string().openapi({ example: 'Design único e formato inovador' }),
  price: z.number().positive().openapi({ description: 'Número maior que zero' }),
  quantity: z.number().positive().openapi({ description: 'Número maior que zero' }),
  image: z.string().url().openapi({ description: 'Uma URL válida' })
});
```

Documentando uma API

Usando a zod-to-openapi - Rotas

```
registry.registerPath({
  method: "get",
  path: "/products",
  description: "Retorna todos os produtos",
  responses: {
    200: {
      description: "Lista de produtos",
      content: {
        "application/json": {
          schema: productsResponseSchema
        }
      }
    },
  },
  tags: ["Products"]
})
```

Documentando uma API

Usando a zod-to-openapi - Gerador

```
import { OpenAPIRegistry, OpenApiGeneratorV3 } from "@asteasolutions/zod-to-
openapi";

export const registry = new OpenAPIRegistry();
export function buildOpenAPIDocument() {
  const generator = new OpenApiGeneratorV3(registry.definitions);
  const doc = generator.generateDocument({
    openapi: '3.0.0',
    info: {
      version: '1.0.0',
      title: 'Products API',
      description: 'API for managing products',
    },
    servers: [{ url: 'http://localhost:3000' }],
  })
  return doc;
}
```

Documentando uma API

Usando a zod-to-openapi - Configurando o server

```
import { buildOpenAPIDocument } from "./docs/openapi.js";

const app = createApp();

// build doc (gera a partir do registry preenchido pelas rotas)
const openapiDoc = buildOpenAPIDocument();

// serve swagger em /docs
app.use("/docs", swaggerUi.serve, swaggerUi.setup(openapiDoc));
```

Referências

- Project structure for an Express REST API when there is no "standard way"
- How to structure an Express.js REST API with best practices
- How to create a REST API with Node.js and Express
- REST API Design Rulebook, Mark Masse
- Designing API responses
- REST API Best Practices
- Zod

Referências

- Schema validation in TypeScript with Zod
- A Complete Guide to Zod
- What Is OpenAPI?

Por hoje é só