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
- TypeORM
- Versionamento
- Paginação
- Filtragem e Ordenação

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

TypeORM

TypeORM

O que é um ORM?

- ORM - *Object Relational* Mapper
  - Software criado para funcionar como ponte entre a representação de dados em:
    - Um banco dados relacional (tabelas, tuplas)
    - Programas orientados a objetos (Classes, instâncias)
  - Naturalmente elas não se integram de forma simples
  - Permite a interação com um banco de dados relacional usando programação OO
  - Manipulação dos dados como se fossem objetos, sem precisar escrever SQL

TypeORM

O que é um ORM?

- Serve como camada de abstração entre a aplicação e o banco de dados
- Prometem um aumento de produtividade
  - Evitando código boilerplate
  - Técnicas que possam ser não ergonômicas ou idiomáticas

TypeORM

Vantagens

- Aumenta a velocidade do time de desenvolvimento
- Diminui o custo de desenvolvimento
- Abstrai a lógica de interação com o banco de dados
- Mais seguro
- Portabilidade

TypeORM

Desvantagens

- Custo de aprendizado do próprio ORM
- Desempenho
  - Em geral, são mais lentos que códigos que usam SQL diretamente

TypeORM

O que é?

- É um ORM para Node.js que suporta TypeScript e JavaScript (ES6)
- Facilita:
  - A definição de modelos de dados,
  - A realização de consultas
  - A execução de operações CRUD
- Fortemente inspirado em ORMs com Hibernate, EntityFramework e Doctrine

TypeORM

Conceitos básicos

- Entidades:
  - Classes que representam tabelas no banco de dados
  - Cada instância da classe representa uma linha da tabela
- Atributos:
  - Propriedades da entidade que mapeiam para colunas da tabela
- Relacionamentos:
  - Definem como as entidades se relacionam entre si, por exemplo, 1:1, 1:N, N:M

TypeORM

Instalação das dependências necessárias

```
npm install typeorm reflect-metadata sqlite3
```

TypeORM

Configurando o tsconfig

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

TypeORM

Criando o datasource

```
import "reflect-metadata";
import { DataSource } from "typeorm";
import { User } from "./entity/User"; // Import your entities

export const AppDataSource = new DataSource({
    type: "sqlite",
    database: "database.sqlite", // The file path for your database
    entities: [User], // List of entities
    synchronize: true, // Automatically create database schema (use
migrations in production)
    logging: false, // Set to true to log SQL queries
    migrations: ["src/migrations/*.ts"]
});
```

TypeORM

Entidade

```
import { OneToMany } from "typeorm";
import { Post } from "./Post";

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    username: string;

    @Column()
    email: string;

    @Column()
    passwordHash: string;

    @Column()
    fullName: string;

    @Column({ default: true })
    isActive: boolean;

    @OneToMany(() => Post, post => post.user)
    posts: Post[];
}
```

Define uma entidade

Define a chave primária

Mapeia o atributo username
para o atributo username da tabela

Atributo que possui um valor padrão

Define um relacionamento entre a tabela User e Post. Com chave estrangeira em post.user

TypeORM

Repositório

```
import { AppDataSource } from "../config/data-source";
import { Product } from "../entities/Product";

export const ProductRepository =
AppDataSource.getRepository(Product).extend({
  findByName(name: string) {
    return this.findOneBy({ name });
  }
});
```

TypeORM

Service

```
import { ProductRepository } from "../repositories/product.repository";
import { Product } from "../entities/Product";

export class ProductService {
  async listAll() {
    return ProductRepository.find();
  }

  async getById(id: number) {
    const product = await ProductRepository.findOneBy({ id });
    if (!product) throw new Error("PRODUCT_NOT_FOUND");
    return product;
  }
```

TypeORM

Configurando o server

```
import express from "express";
import { AppDataSource } from "./config/data-source";

import { productRoutes } from "./modules/products/product.routes";

const app = express();
app.use(express.json());

// Inicialização do banco
AppDataSource.initialize()
  .then(() => console.log("📦 Database connected"))
  .catch((err) => console.error("❌ Error connecting database:", err));

// Rotas da API
app.use("/api/v1/products", productRoutes);

export default app;
```

Versionamento

Versionamento

! O problema: APIs evoluem

- Requisitos mudam
- Regras de negócio evoluem
- Campos são adicionados, removidos ou alterados
- Formatos de resposta podem mudar
- Sem versionamento, qualquer mudança pode quebrar clientes existentes

Versionamento

O que significa versionar uma API?

- Manter múltiplas versões da mesma API em funcionamento
- Permitir que clientes antigos continuem funcionando
- Dar liberdade para evolução controlada da API

Versionamento

🤔 Quando o versionamento é necessário?

- Mudança no formato da resposta
- Alteração de significado de um campo
- Remoção de campos
- Mudanças incompatíveis com versões anteriores (breaking changes)
  - Ex: Transformar um campo opcional em obrigatório

Versionamento

Estratégias de versionamento

- Versionamento na URL
- Versionamento via Header
- Versionamento via Query Param
- Versionamento baseado em Content Negotiation (Accept Header)

Versionamento

1️⃣ Versionamento na URL

```
GET /api/v1/users
GET /api/v2/users
```

- Vantagens
  - Simples de entender
  - Fácil de testar
  - Muito comum no mercado
- Desvantagens
  - A URL muda a cada versão
  - Alguns puristas argumentam que a URL representa o recurso, não a versão

Versionamento

2️⃣ Versionamento via Header

- Vantagens
  - URL permanece limpa
  - Mais alinhado com princípios REST
- Desvantagens
  - Menos visível
  - Mais difícil de testar manualmente
  - Exige maior maturidade da equipe
  - Duplicação do comportamento do Accept

```
GET /api/users
Header: X-API-Version: 1
```

Versionamento

3️⃣ Versionamento via Query Param

- Vantagens
  - Fácil de implementar
  - Simples de testar
- Desvantagens
  - Pode gerar confusão
  - Não é considerado uma boa prática consolidada

```
GET /api/users?version=1
```

Versionamento

4️⃣ Content Negotiation

- Vantagens

```
Accept: application/vnd.myapi.v1+json
```

  - Muito elegante
  - Bastante aderente ao REST
- Desvantagens
  - Complexo
    - Os clientes precisam saber quais cabeçalhos usar antes de solicitar um recurso
  - Overkill para a maioria dos projetos

Versionamento

Boas práticas

- Projete pensando em extensibilidade
- Garanta compatibilidade retroativa
- Documentar todas as versões
- Evitar alterar endpoints e respostas
- Conheça seus consumidores
- Defina uma política de versionamento
- Publique e mantenha um cronograma de releases

Versionamento

1️⃣ Projete pensando em extensibilidade

- Algumas decisões dificultam evolução
- Cuidados comuns:
  - Tipos como booleanos e arrays simples são mais frágeis
  - Prefira estruturas extensíveis (objetos, enums bem pensados)
- Pense no “futuro” do dado desde a v1

👉 Um bom design inicial reduz a necessidade de novas versões

Versionamento

2️⃣ Garanta compatibilidade retroativa (Backward Compatibility)

- Sempre que possível, não quebre clientes existentes
  - Mudanças forçam consumidores a atualizar seus sistemas, muitas quebras em pouco tempo reduzem a confiança na API
  - APIs estáveis são mais adotadas e mantidas por mais tempo
- Boas práticas:
  - Usar testes automatizados para garantir que schemas de request/response não mudaram
  - Integrar esses testes ao CI para evitar releases quebrados
- 👉 Compatibilidade reduz retrabalho, documentação extra e suporte ao cliente.

Versionamento

3️⃣ Documente todas as versões

- Nunca documente apenas a última versão da API
  - Clientes podem estar usando versões antigas
  - Falta de documentação dificulta manutenção e migração
- Boas práticas:
  - Documentar cada versão separadamente
  - Manter um changelog claro explicando o que mudou
  - Oferecer canais de atualização (e-mail, RSS, release notes)

👉 Changelogs ajudam consumidores a decidir se e quando migrar

Versionamento

4️⃣ Evite mudar endpoints ou formatos de resposta

- Alterar endpoints ou respostas existentes é uma das maiores causas de breaking changes
- Prefira:
  - Criar novos endpoints
  - Adicionar novos campos (em vez de alterar/remover os antigos)
- Exemplo:

  ❌ Mudar phone para array

  ✅ Manter phone e adicionar phones

👉 APIs devem evoluir por adição, não por modificação destrutiva

Versionamento

5️⃣ Conheça seus consumidores

- Nem todo uso da API é explícito
  - Clientes podem depender de comportamentos não documentados
  - Contrato invisível
- Exemplo:
  - Cliente acessa propriedades por índice em vez de nome

👉 Antes de mudar algo, entenda como a API é realmente usada

Versionamento

6️⃣ Defina uma política clara de versionamento

- Especialmente em APIs públicas ou monetizadas
- Inclua nos termos:
  - O que é considerado breaking change
  - Como e quando os clientes serão avisados
  - Quanto tempo terão para migrar

👉 Transparência aumenta confiança e adoção

Versionamento

7️⃣ Publique e mantenha um cronograma de releases

- Consumidores precisam saber o que vai mudar e quando
- Boas práticas:
  - Informar datas de lançamento
  - Avisar com antecedência sobre depreciações
  - Monitorar uso das versões antigas
  - Não remover versões ainda muito utilizadas

👉 Um bom cronograma evita downtime e perda de receita para clientes

Paginação

Paginação

Introdução

- Refere-se a uma técnica utilizada no design e desenvolvimento de APIs para recuperar grandes conjuntos de dados em páginas menores
  - Cada página contém um número limitado de registros ou entradas
  - O cliente da API pode então solicitar páginas subsequentes para recuperar dados adicionais

Paginação

O problema de não paginar

- Uma requisição GET pode retorna milhares de registros
- Degradação da performance do backend
- Frontend difícil de renderizar => Paginação no front
- Escalabilidade comprometida

Paginação

Vantagens de paginar

🚀 Melhor desempenho

💾 Uso eficiente de recursos

🙂 Melhor experiência do usuário

🌐 Transferência de dados otimizada

📈 Escalabilidade e flexibilidade

🛠️ Tratamento de erros mais simples

Paginação

🚀 Melhor desempenho

- A paginação reduz o tempo de resposta
- Diminui o esforço de processamento no servidor
- Acelerando o consumo no cliente

Paginação

💾 Uso eficiente de recursos

- Ao evitar o carregamento de grandes volumes de dados de uma só vez, a paginação reduz:
  - Consumo de memória
  - Uso de CPU
  - Tráfego de rede
- Isso melhora a escalabilidade e pode até reduzir custos de infraestrutura

Paginação

🙂 Melhor experiência do usuário

- Os dados são entregues de forma progressiva e controlada, permitindo:
  - Carregamento mais rápido
  - Interfaces mais responsivas
  - Navegação mais simples em grandes listas

Paginação

🌐 Transferência de dados otimizada

- Somente os dados realmente necessários são enviados pela rede, o que:
  - Reduz o consumo de banda
  - Melhora o desempenho em conexões lentas
  - Evita desperdício de dados

Paginação

📈 Escalabilidade e flexibilidade

- A paginação permite que a API cresça junto com os dados, suportando:
  - Grandes volumes de registros
  - Múltiplos dispositivos
  - Diferentes padrões de consumo

Paginação

🛠️ Tratamento de erros mais simples

- Em caso de falha:
  - apenas a página afetada precisa ser reprocessada
  - não é necessário recarregar todo o conjunto de dados
- Isso torna o sistema mais robusto e resiliente

Paginação

Estratégias de paginação

- Offset-based pagination
- Page-based pagination
- Keyset pagination
- Time-based pagination
- Cursor-based

Paginação

Offset-based pagination

- Método mais direto
- Utiliza dois parâmetros: LIMIT e OFFSET
- Implementação direta quando se usa bancos de dados SQL

```
GET /users?offset=20&limit=10
```

```
SELECT * FROM users
LIMIT 10 OFFSET 20
```

Paginação

Offset-based pagination

✅ Vantagens

- Muito simples de entender
- Fácil de implementar
- Compatível com quase todos os bancos e ORMs
- Bom pequenas bases de dados

❌ Desvantagens

- Baixa performance em grandes volumes
  - O banco precisa escanear cada linha para realizar o salto
- Resultados inconsistentes se dados forem inseridos/removidos

Paginação

Offset-based pagination

🎯 Quando usar

- APIs internas
- Listagens pequenas
- Casos didáticos iniciais

Paginação

Page-based pagination

- Divide os dados em páginas de forma uniforme
- Permite que o tamanho da página seja escolhido
- O servidor calcula o offset

```
GET /users?page=3&limit=10
// offset = (page - 1) * limit
```

Paginação

Page-based pagination

✅ Vantagens

- Intuitiva para frontend e UX
- Elimina a chance de acessar uma página não existente
- Compatível com UI tradicional (páginas numeradas)

❌ Desvantagens

- Herda todos os problemas do offset-based

Paginação

Page-based pagination

🎯 Quando usar

- APIs públicas simples
- Dashboards administrativos
- Projetos educacionais

Paginação

Keyset pagination

- Baseada em um campo ordenado
  - Valor único
- Usado em tabelas grandes ou com dados que mudam com frequência

```
GET /users?afterId=100&limit=10
```

```
SELECT * FROM users
WHERE id > 100
ORDER BY id ASC
LIMIT 10
```

Paginação

Keyset pagination

✅ Vantagens

- Muito performática
- Resultados estáveis
- Ideal para grandes tabelas

❌ Desvantagens

- Depende de ordenação consistente
- Não suporta navegação aleatória
- Funciona melhor apenas para “próxima página”

Paginação

Keyset pagination

🎯 Quando usar

- APIs REST com grandes volumes
- Logs, eventos, histórico

Paginação

Time-based pagination

- Baseada em um critério de tempo
- Utiliza o timestamp para dividir e recuperar os registros
- Em geral, o cliente especifica o intervalo de tempo

```
GET /events?
after=2025-01-01T10:00:00Z&limit=10
```

```
SELECT * FROM users
WHERE createdAt >
2025-01-01T10:00:00
ORDER BY id createdAt
LIMIT 10
```

Paginação

Time-based pagination

✅ Vantagens

- Excelente para dados temporais
- Funciona bem com streams e eventos
- Escala bem

❌ Desvantagens

- Pode haver conflitos se timestamps não forem únicos
- Depende de relógios consistentes
- Não é genérica

Paginação

Time-based pagination

🎯 Quando usar

- Logs
- Eventos
- Sistemas de monitoramento

Paginação

Cursor-based pagination

- A cada requisição os dados são retornados juntamente com um cursor
  - Um marcador que destaca um item específico no conjunto de dados
- Pode ser baseado em vários critérios:
  - timestamp, chave primária ou uma representação codificada do registro
  - Ex: Slack codifica informações em base64 usando nome de campo e seu valor, bem como uma ordem. String opaca: dXNlcjpXMDdRQ1JQQTQ=.

```
GET /users?cursor=dXNlcjpXMDdRQ1JQQTQ&limit=10
```

Paginação

Cursor-based pagination

✅ Vantagens

- Alta performance
- Resultados consistentes
- Ideal para feeds e scroll infinito
- Escala muito bem
- A lógica interna pode ser alterada sem impactar no código do cliente

❌ Desvantagens

- Mais difícil de implementar
- Não permite pular páginas arbitrariamente
- Mais difícil de depurar

Paginação

Time-based pagination

🎯 Quando usar

- Feeds (Instagram, Twitter)
- APIs de alto tráfego
- Grandes volumes de dados

Paginação

Time-based pagination

Estratégia	Parâmetros	Performance	Consistência	Facilidade	Escala bem	Navegação aleatória
Offset-based	offset + limit	❌ Baixa	❌ Baixa	✅ Alta	❌	✅
Page-based	page + limit	❌ Baixa	❌ Baixa	✅ Alta	❌	✅
Keyset	afterId + limit	✅ Alta	✅ Alta	⚠️ Média	✅	❌
Time-based	timestamp + limit	✅ Alta	⚠️ Média	⚠️ Média	✅	❌
Cursor-based	cursor + limit	✅ Alta	✅ Alta	❌ Baixa	✅	❌

Filtragem e Ordenação

Referências

- What is an ORM?
- What is an ORM – The Meaning of Object Relational Mapping Database Tools
- How to Version REST APIs: A Comprehensive Guide
- API Versioning: Strategies & Best Practices
- REST API Versioning: How to Version a REST API?
- API Design Basics: Pagination
- What is REST API pagination?

Referências

- Unlocking the Power of API Pagination: Best Practices and Strategies
- REST API Design: Filtering, Sorting, and Pagination
- REST API Response Pagination, Sorting and Filtering
- Filtering Collections

Por hoje é só