MVC não é o padrão mais eficiente para APIs

Apesar de ter sido amplamente utilizado por décadas, o padrão MVC (Model-View-Controller) não é o mais indicado para desenvolvimento de APIs modernas. MVC surgiu no contexto de aplicações com interface gráfica (GUIs), onde a separação entre View e Controller fazia mais sentido. Em APIs, especialmente RESTful ou GraphQL, não há "View" no sentido tradicional, o que faz com que o padrão fique desbalanceado ou adaptado de forma forçada.

Alternativas como ADR (Action-Domain-Responder), Clean Architecture, Hexagonal Architecture (Ports and Adapters) e CQRS (Command Query Responsibility Segregation) oferecem melhor desacoplamento, maior coesão e testabilidade mais robusta — aspectos cruciais para APIs escaláveis e de longo prazo.

Por que MVC falha em APIs?

1. **Acoplamento entre camadas**
   Controladores em MVC frequentemente manipulam lógica de negócio e detalhes de resposta HTTP ao mesmo tempo. Isso fere o princípio da *responsabilidade única* (SRP) e dificulta testes unitários.
2. **Distorção do papel do "View"**
   Em APIs, a *view* geralmente é um JSON gerado a partir de dados. Assim, não faz sentido manter uma camada *view* separada — ela não tem comportamento próprio, apenas representa o estado.
3. **Dificuldade em organizar domínios complexos**
   Em aplicações de múltiplos domínios, o MVC tende a gerar estruturas planas e desorganizadas. Controllers incham com validações, regras de negócio e lógica de resposta. Isso compromete a manutenibilidade.
4. **Baixa testabilidade**
   Com responsabilidades misturadas, testes unitários se tornam difíceis. Para testar regras de negócio, muitas vezes é necessário mockar partes da infraestrutura HTTP — algo que deveria estar separado.

Estrutura proposta

Inspirada em **Clean Architecture** e **ADR**, a estrutura a seguir é altamente modular, testável e escalável:

```
src/
├── api/                          # Rotas da API (Next.js API Routes)
│   ├── users/                    # Exemplo: recurso "users"
│   │   ├── [id].ts               # Rota dinâmica (ex: /api/users/1)
│   │   └── index.ts              # Rota base (ex: /api/users)
│   └── ...                       # Outros recursos (products, auth,...)
│
├── app/                          # Lógica de aplicação
│   ├── modules/                  # Módulos por domínio (ex: users)
│   │   ├── users/                # Módulo "users"
│   │   │   ├── dtos/             # Data Transfer Objects (DTOs)
│   │   │   │   └── user.dto.ts   # Definição de formatos de entrada/saída
│   │   │   ├── services/         # Lógica de negócio
│   │   │   │   └── user.service.ts
│   │   │   ├── repositories/     # Acesso a dados (banco, cache,...)
│   │   │   │   └── user.repository.ts
│   │   │   └── types/            # Tipos TypeScript
│   │   │       └── user.types.ts
│   │   └── ...                   # Outros módulos
│   │
│   ├── core/                     # Configurações globais
│   │   ├── database/             # Conexão com o banco (ex: Prisma)
│   │   ├── errors/               # Erros customizados (ex: NotFoundError)
│   │   └── middleware/           # Middlewares globais (ex: auth)
│   │
│   └── utils/                    # Utilitários (validação, logging, etc.)
│
└── tests/                        # Testes (Jest/Vitest)
    ├── unit/                     # Testes de unidades (services)
    └── integration/              # Testes de integração (API)
```

Fluxo de uma Requisição (Resumo)

1. **API Handler**
   Lida com HTTP e validações iniciais.
2. **Service**
   Contém a regra de negócio pura.
3. **Repository**
   Se comunica com banco de dados, cache ou serviços externos.
4. **DTO**
   Define contratos de entrada/saída, evitando vazamento de detalhes internos.

Fluxo de uma Requisição (Detalhado)

1. **Rota (**/api/users/[id].ts**)**
    - Recebe a requisição HTTP.
    - Valida dados de entrada (ex: com Zod).
    - Chama o **Service** correspondente.

```
// api/users/[id].ts
import { NextApiRequest, NextApiResponse } from 'next';
import { getUserById } from '@/app/modules/users/services/user.service';
import { NotFoundError } from '@/app/core/errors';

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  try {
    const { id } = req.query;
    const user = await getUserById(id as string);

    if (!user) throw new NotFoundError('User not found');

    res.status(200).json(user);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
}
```

2. **Service (**user.service.ts**)**
    - Contém a lógica de negócio.
    - Usa o Repository para interagir com o banco.

```
// app/modules/users/services/user.service.ts
import { getUserById as getUserByIdRepo } from '../repositories/user.repository';
import { UserResponseDto } from '../dtos/user.dto';

export const getUserById = async (id: string): Promise<UserResponseDto> => {
  const user = await getUserByIdRepo(id);
  return {
    id: user.id,
    name: user.name,
    email: user.email, // DTO filtra dados sensíveis (ex: senha)
  };
};
```

3. **Repository (**user.repository.ts**)**
    - Camada pura de **acesso a dados** (banco, API externa, cache).

```
// app/modules/users/repositories/user.repository.ts
import prisma from '@/app/core/database/prisma';

export const getUserByIdRepo = async (id: string) => {
  return await prisma.user.findUnique({ where: { id } });
};
```

4. **DTO (**user.dto.ts**)**
    - Define **formatos** de entrada/saída para evitar vazamento de dados.

```
// app/modules/users/dtos/user.dto.ts
export interface UserResponseDto {
  id: string;
  name: string;
  email: string;
}
```

Comparação Prática: MVC vs Clean Architecture

Aspecto	**MVC Tradicional**	**Clean Architecture (Proposta)**
Separação de responsabilidades	Baixa	Alta
Testabilidade	Limitada	Excelente (camadas isoladas)
Escalabilidade	Difícil com o tempo	Natural via separação por domínio
Substituição de tecnologia	Acoplada	Desacoplada (ex: banco, framework)
Clareza no fluxo	Confusa em grandes apps	Clara e orientada a casos de uso

Vantagens dessa Estrutura

Separação clara de responsabilidades:

- Rotas: lidam com HTTP.
- Services: regras de negócio.
- Repositories: acesso a dados.

Testabilidade:

- Services e repositories podem ser mockados facilmente.

Flexibilidade:

- Trocar o banco de dados? Basta alterar o repository.

- Mudar de Next.js para Express? A lógica de negócio (services) permanece.

Segurança:

- DTOs evitam expor dados sensíveis (ex: senhas).

Conceitos Fundamentais

Clean Architecture

Conceito proposto por Robert C. Martin ("Uncle Bob"). Define círculos concêntricos onde **as regras de negócio estão no centro**, isoladas de frameworks, bancos de dados ou qualquer infraestrutura externa.

- Independente de UI, banco, framework
- Inversão de dependências
- Alta testabilidade

ADR (Action-Domain-Responder)

É um refinamento para APIs RESTful:

- **Action**: recebe e roteia a requisição
- **Domain**: executa lógica de negócio (services)
- **Responder**: gera resposta HTTP (normalmente embutido no handler em Next.js)

Conclusão

Enquanto o padrão MVC ainda pode ter seu lugar em aplicações front-end com UI complexa, **para APIs modernas o seu uso é desaconselhado**. Arquiteturas como Clean Architecture ou ADR **promovem melhor organização, menor acoplamento, alta coesão e flexibilidade de longo prazo**, especialmente em sistemas distribuídos, microsserviços ou backends desacoplados.

Adotar essas práticas é um passo decisivo rumo a APIs sustentáveis e de alta qualidade.