7 - Coding Standard - Backend

Iagro Backend

Backend da aplicação iAgro construído com NestJS, seguindo princípios de Domain-Driven Design (DDD) e Clean Architecture.

🏗️ Estrutura do Projeto

O projeto está organizado seguindo os princípios de Clean Architecture e DDD, com uma estrutura modular bem definida:

```
src/
├── modules/           # Módulos de negócio
│   ├── health/       # Módulo de saúde da aplicação
│   └── tenants/      # Módulo de inquilinos/tenants
├── shared/           # Componentes compartilhados
│   ├── domain/       # Conceitos de domínio compartilhados
│   ├── infrastructure/ # Infraestrutura compartilhada
│   ├── application/  # Abstrações de aplicação
│   └── interfaces/   # Interfaces e DTOs compartilhados
└── main.ts           # Ponto de entrada da aplicação
```

📁 Módulos de Negócio

Cada módulo segue a mesma estrutura interna:

```
modules/tenants/
├── domain/           # Camada de domínio
│   ├── entities/     # Entidades de domínio
│   ├── repositories/ # Interfaces de repositório
│   └── value-objects/ # Objetos de valor
├── application/      # Camada de aplicação
│   ├── services/     # Serviços de aplicação
│   ├── use-cases/    # Casos de uso
│   └── dto/          # DTOs de apresentação
├── infrastructure/   # Camada de infraestrutura
│   ├── repositories/ # Implementações de repositório
│   └── mappers/      # Mapeadores de dados
└── interfaces/       # Camada de interface
    └── rest/         # Controllers REST
```

> **Nota**: nem todos os módulos seguem 100% deste template. Em módulos mais recentes (ex.: `smart-caldas`) os DTOs de request/response ficam em `interfaces/rest/dto/{request,response}` em vez de `application/dto`; e em vez de `domain/value-objects` alguns módulos usam `domain/constants` ou `domain/events`. Também é comum haver subpastas extras como `infrastructure/interceptors` ou `interfaces/websocket` quando o módulo expõe WebSocket.

🔧 Componentes Compartilhados

`/shared/domain`

- **Entidades base**: Classes abstratas para entidades e agregados
- **Objetos de valor**: Identificadores e outros objetos de valor
- **Repositórios**: Interfaces base para repositórios

`/shared/infrastructure`

- **Mapeadores**: Utilitários para transformação de dados
- **Prisma**: Serviço de banco de dados

`/shared/application`

- **Casos de uso**: Interface base para casos de uso

🎯 Conceitos de DDD (Domain-Driven Design)

Entidades (Entities)

- Representam objetos com identidade única
- Podem ter estado mutável
- Exemplo: `Tenant`, `User`, `Product`

Objetos de Valor (Value Objects)

- Representam conceitos imutáveis
- Não possuem identidade própria
- Exemplo: `Email`, `Money`, `Address`

Agregados (Aggregates)

- Conjunto de entidades e objetos de valor relacionados
- Possuem uma raiz de agregado

- Garantem consistência transacional

Repositórios (Repositories)

- Abstraem a persistência de entidades
- Fornecem interface para busca e armazenamento
- Escondem detalhes de implementação do banco
- ⚠️ **IMPORTANTE**: Os repositórios devem apenas persistir e buscar entidades. Todas as operações de atualização, validação e lógica de negócio devem ser implementadas nas próprias entidades de domínio, não nos repositórios.

Casos de Uso (Use Cases)

- Representam operações de negócio específicas
- Orquestram entidades e objetos de valor
- Implementam regras de negócio

🗄️ Convenções de Banco de Dados

Estrutura Padrão das Tabelas

Todas as tabelas devem conter os seguintes campos obrigatórios:

- `id` : Identificador único (UUID string ou número incremental)
- `created_at` : Data de criação do registro
- `updated_at` : Data da última atualização
- `deleted_at` : Data de exclusão lógica (soft delete)
- `disabled_at` : Data de desabilitação do registro

Convenções de Nomenclatura

- **Banco de dados**: Use `snake_case` para tabelas e colunas
- **Código TypeScript**: Use `camelCase` para propriedades e métodos
- **Mapeamento**: Configure o Prisma para mapear automaticamente

Exemplo de Schema Prisma

```
model User {
  id          String    @id @default(uuid())
  name        String
  email       String    @unique
  createdAt   DateTime  @default(now()) @map("created_at")
  updatedAt   DateTime  @updatedAt @map("updated_at")
  deletedAt   DateTime? @map("deleted_at")
  disabledAt  DateTime? @map("disabled_at")

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

⚠️ Importante: Criar Modelos Prisma Primeiro

Antes de implementar o código, sempre crie/atualize os modelos no schema do Prisma (`prisma/schema.prisma`). Isso garante que:

- As tabelas sejam criadas com a estrutura correta
- Os campos obrigatórios estejam presentes
- O mapeamento snake_case ↔ camelCase esteja configurado
- As migrações sejam geradas corretamente

🐳 Executando o PostgreSQL com Docker

Para desenvolvimento local, use o Docker Compose incluído no projeto:

```
# Subir o PostgreSQL
docker compose up -d postgres

# Verificar status
docker compose ps

# Parar os serviços
docker compose down
```

O arquivo `docker-compose.yml` já está configurado com as variáveis de ambiente necessárias.

🚀 Como Adicionar um Novo Módulo

0. Usar o Script de Criação Automática

Para facilitar a criação de novos módulos, use o script incluído no projeto:

```
# Criar um novo módulo com toda a estrutura necessária
pnpm create:module nome-do-modulo

# Exemplo:
pnpm create:module users
```

Este script criará automaticamente:

- ✅ Estrutura de pastas completa seguindo Clean Architecture
- ✅ Todos os diretórios necessários (domain, application, infrastructure, interfaces)
- ⚠️ Arquivos `index.ts` apenas nas 4 pastas de topo (`domain/` , `application/` , `infrastructure/` , `interfaces/`) e no `index.ts` raiz do módulo — não gera `index.ts` dentro das subpastas (`domain/entities` , `application/services` etc.)
- ✅ Estrutura pronta para implementação
- ⚠️ Requer shell POSIX (usa `mkdir -p` / `touch` via `execSync`) — em Windows precisa rodar via Git Bash/WSL, não funciona em cmd.exe puro

1. Criar/Atualizar Modelos Prisma

⚠️ **IMPORTANTE**: Sempre comece criando/atualizando os modelos no `prisma/schema.prisma` antes de implementar o código:

```
model NewModule {
  id          String    @id @default(uuid())
  name        String
  description String?
  createdAt   DateTime  @default(now()) @map("created_at")
  updatedAt   DateTime  @updatedAt @map("updated_at")
  deletedAt   DateTime? @map("deleted_at")
  disabledAt  DateTime? @map("disabled_at")

  @@map("new_modules")
}
```

2. Criar a Estrutura de Pastas

```
mkdir -p src/modules/new-module/{domain/{entities,repositories,value-objects},application/{ser
```

2. Definir a Entidade de Domínio

```
// src/modules/new-module/domain/entities/new-module.domain.ts
import { TenantlessAggregateRoot } from 'src/shared/domain';
import { NewModuleIdentifier } from './new-module.identifier';

export interface NewModuleProps {
  name: string;
  description?: string;
  createdAt?: Date;
}

export class NewModule extends TenantlessAggregateRoot<
  NewModuleProps,
  NewModuleIdentifier
> {
  get name() {
    return this.props.name;
  }

  get description() {
    return this.props.description;
  }

  get createdAt() {
    return this.props.createdAt;
  }

  static create(props: NewModuleProps, id?: NewModuleIdentifier): NewModule {
    return new NewModule(props, id);
  }
}
```

4. Criar o Identificador

```
// src/modules/new-module/domain/entities/new-module.identifier.ts
import { Identifier } from 'src/shared/domain';

export class NewModuleIdentifier extends Identifier<string> {
  constructor(id: string) {
    super(id);
  }
```

```
}
```

5. Definir a Interface do Repositório

```typescript
// src/modules/new-module/domain/repositories/new-module.repository.ts
import { TenantlessRepository } from 'src/shared/domain';
import { NewModule } from '../entities/new-module.domain';

export interface INewModuleRepository extends TenantlessRepository<NewModule> {
  findByName(name: string): Promise<NewModule | null>;
}
```

6. Criar o Caso de Uso

```typescript
// src/modules/new-module/application/use-cases/create-new-module.use-case.ts
import { UseCase } from 'src/shared/application';
import { NewModule } from '../../domain/entities/new-module.domain';
import { INewModuleRepository } from '../../domain/repositories/new-module.repository';

export interface CreateNewModuleRequest {
  name: string;
  description?: string;
}

export interface CreateNewModuleResponse {
  id: string;
  name: string;
  description?: string;
  createdAt: Date;
}

export class CreateNewModuleUseCase
  implements UseCase<CreateNewModuleRequest, CreateNewModuleResponse>
{
  constructor(private readonly repository: INewModuleRepository) {}

  async execute(
    request: CreateNewModuleRequest,
  ): Promise<CreateNewModuleResponse> {
    const newModule = NewModule.create({
      name: request.name,
      description: request.description,
      createdAt: new Date(),
    });

    const saved = await this.repository.persist(newModule);

    return {
      id: saved.id!.value,
      name: saved.name,
      description: saved.description,
      createdAt: saved.createdAt!,
    };
  }
}
```

7. Implementar o Repositório

```typescript
// src/modules/new-module/infrastructure/repositories/new-module.repository.impl.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from 'src/shared/infrastructure';
import { NewModule } from '../../domain/entities/new-module.domain';
import { NewModuleIdentifier } from '../../domain/entities/new-module.identifier';
import { INewModuleRepository } from '../../domain/repositories/new-module.repository';
```

```typescript

@Injectable()
export class NewModuleRepositoryImpl implements INewModuleRepository {
  constructor(private readonly prisma: PrismaService) {}

  async persist(entity: NewModule): Promise<NewModule> {
    const data = {
      id: entity.id?.value,
      name: entity.name,
      description: entity.description,
      createdAt: entity.createdAt,
    };

    const saved = await this.prisma.newModule.upsert({
      where: { id: data.id || 'new' },
      create: data,
      update: data,
    });

    return NewModule.create(
      {
        name: saved.name,
        description: saved.description,
        createdAt: saved.createdAt,
      },
      new NewModuleIdentifier(saved.id),
    );
  }

  async findById(id: NewModuleIdentifier): Promise<NewModule | null> {
    const found = await this.prisma.newModule.findUnique({
      where: { id: id.value },
    });

    if (!found) return null;

    return NewModule.create(
      {
        name: found.name,
        description: found.description,
        createdAt: found.createdAt,
      },
      new NewModuleIdentifier(found.id),
    );
  }

  async findByName(name: string): Promise<NewModule | null> {
    const found = await this.prisma.newModule.findFirst({
      where: { name },
    });

    if (!found) return null;

    return NewModule.create(
      {
        name: found.name,
        description: found.description,
        createdAt: found.createdAt,
      },
      new NewModuleIdentifier(found.id),
    );
  }

  async findAll(): Promise<NewModule[]> {
    const all = await this.prisma.newModule.findMany();

    return all.map((item) =>
```

```
      NewModule.create(
        {
          name: item.name,
          description: item.description,
          createdAt: item.createdAt,
        },
        new NewModuleIdentifier(item.id),
      ),
    );
  }

  async deleteById(id: NewModuleIdentifier): Promise<void> {
    await this.prisma.newModule.delete({
      where: { id: id.value },
    });
  }
}
```

8. Criar o Controller

```
// src/modules/new-module/interfaces/rest/controllers/new-module.controller.ts
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  Post,
  Put,
} from '@nestjs/common';
import { CreateNewModuleUseCase } from '../../../application/use-cases/create-new-module.use-

@Controller('new-module')
export class NewModuleController {
  constructor(private readonly createUseCase: CreateNewModuleUseCase) {}

  @Post()
  async create(@Body() request: any) {
    return this.createUseCase.execute(request);
  }
}
```

8.1. Configurar ClassProvider no Service

Cada service deve exportar um ClassProvider com as seguintes características:

```
// src/modules/new-module/application/services/create-new-module.service.ts
import { ClassProvider, Injectable } from '@nestjs/common';
import { CreateNewModuleUseCase } from '../use-cases/create-new-module.use-case';

@Injectable()
export class CreateNewModuleService implements CreateNewModuleUseCase {
  // implementação do service
}

export const CREATE_NEW_MODULE_SERVICE: ClassProvider<CreateNewModuleUseCase> =
  {
    provide: CreateNewModuleUseCase,
    useClass: CreateNewModuleService,
  };
```

⚠️ **IMPORTANTE**:

- O nome da constante deve estar em **SCREAMING_SNAKE_CASE**

- O `provide` deve usar o **UseCase** (interface)
- O `useClass` deve usar o **Service** (implementação)
- O mesmo padrão também é usado para providers de repositório (ex.: `LOCATION_REPOSITORY`), não só para services de use-case.

9. Criar o Módulo

```ts
// src/modules/new-module/new-module.module.ts
import { Module } from '@nestjs/common';
import { NewModuleController } from './interfaces/rest/controllers/new-module.controller';
import { CreateNewModuleUseCase } from './application/use-cases/create-new-module.use-case';
import { NewModuleRepositoryImpl } from './infrastructure/repositories/new-module.repository.
import { INewModuleRepository } from './domain/repositories/new-module.repository';

@Module({
  controllers: [NewModuleController],
  providers: [CREATE_NEW_MODULE_SERVICE],
  exports: [INewModuleRepository],
})
export class NewModuleModule {}
```

10. Adicionar ao App Module

```ts
// src/app.module.ts
import { NewModuleModule } from './modules/new-module/new-module.module';

@Module({
  imports: [
    // ... outros módulos
    NewModuleModule,
  ],
})
export class AppModule {}
```

11. Criar a Migração do Prisma

```
npx prisma migrate dev
```

E insira o nome da migration quando solicitado.

🛠️ Tecnologias Utilizadas

- **NestJS**: Framework para construção de aplicações Node.js escaláveis
- **Prisma**: ORM moderno para TypeScript e Node.js
- **TypeScript**: Linguagem de programação tipada
- **PostgreSQL**: Banco de dados relacional

📜 Scripts Disponíveis

Criar Módulo

```
# Cria automaticamente a estrutura completa de um novo módulo
pnpm create:module nome-do-modulo
```

Este script gera automaticamente:

- Estrutura de pastas seguindo Clean Architecture
- Arquivos `index.ts` nas 4 pastas de topo de cada módulo (`domain`, `application`, `infrastructure`, `interfaces`) e no `index.ts` raiz — não em cada subpasta
- Estrutura pronta para implementação
- Requer shell POSIX (Git Bash/WSL) para rodar no Windows

📋 Pré-requisitos

- Node.js 22
- pnpm 10
- Docker

🚀 Instalação e Execução

```
# Instalar dependências
pnpm install

# Configurar variáveis de ambiente
cp .env.example .env

# Subir o PostgreSQL com Docker
docker compose up -d postgres

# Executar migrações
npx prisma migrate dev

# Executar em desenvolvimento
pnpm run start:dev

# Executar testes
pnpm run test

# Executar testes e2e
pnpm run test:e2e
```

📚 Recursos Adicionais

- Documentação do NestJS
- Documentação do Prisma
- Domain-Driven Design
- Clean Architecture

🤝 Contribuição

1. Crie uma task no Jira e anote o ID da task
2. Crie uma branch usando o ID da task (`git checkout -b IAGRO-123`)
3. Faça commits seguindo o padrão convencional:

- `feat: IAGRO-123 add new user authentication system`
- `fix: IAGRO-123 resolve database connection timeout issue`
- `refactor: IAGRO-123 improve error handling in user service`
- `docs: IAGRO-123 update API documentation`
- `test: IAGRO-123 add unit tests for user repository`

4. Push para a branch (`git push origin ABC-123`)
5. Abra um Pull Request referenciando a task do Jira

> **Nota**: este é o padrão recomendado, mas hoje é seguido de forma parcial no histórico do repositório — boa parte dos merges usa o nome da branch como título do PR (ex.: `Feat/iagro-704 (#416)`) em vez do formato `tipo: IAGRO-123 descrição` . Vale reforçar a adesão ao padrão nas revisões de PR.