DOCUMENTAÇÃO DE ARQUITETURA

Gertel Skills Backend

Clean Architecture simplificada — Camadas, Repository Pattern e Use Cases

Node.js · TypeScript · Fastify · Prisma · MariaDB · MongoDB

Sumário

1. Visão Geral

Este projeto utiliza uma **Clean Architecture simplificada** (também chamada de Arquitetura em Camadas com Repository Pattern e Use Cases), inspirada nos princípios de Robert C. Martin (Uncle Bob) e adaptada para a realidade de APIs REST com Node.js/TypeScript.

A ideia central é **separar responsabilidades em camadas**, onde cada camada tem um papel claro e só depende das camadas mais internas — nunca das mais externas.

2. Diagrama de Camadas

Cada camada depende apenas da camada imediatamente mais interna. As setas indicam a direção da dependência: o HTTP conhece os Casos de Uso, que conhecem os Repositórios, que dependem da Infraestrutura.

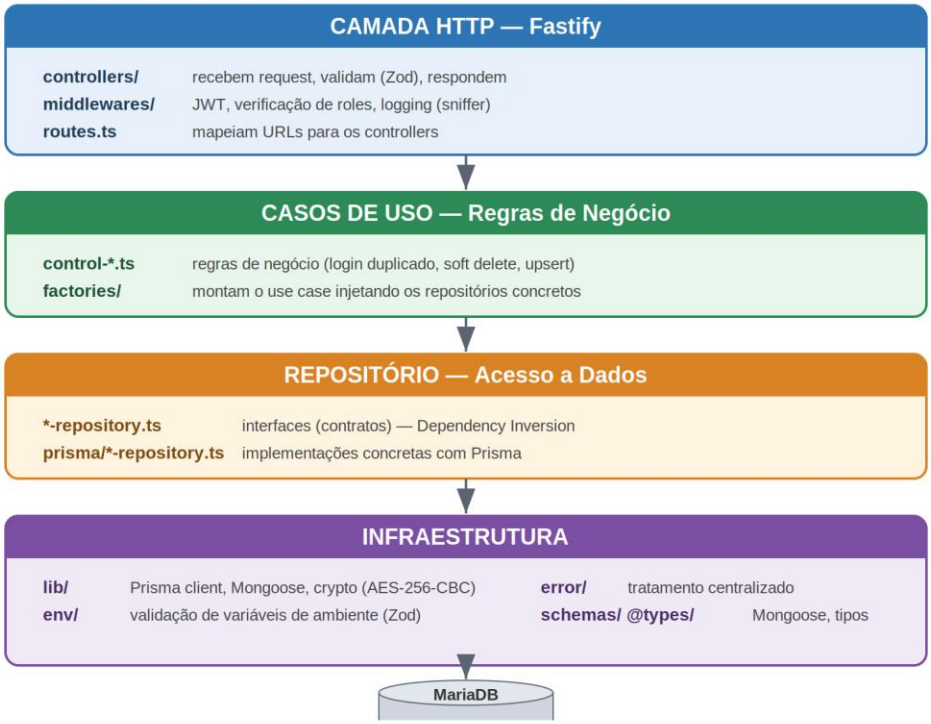

Figura 1 — As quatro camadas da aplicação e o sentido das dependências.

3. Fluxo de uma Requisição

Uma requisição HTTP atravessa as camadas de cima para baixo, sendo progressivamente validada, processada por regras de negócio e finalmente persistida no banco.

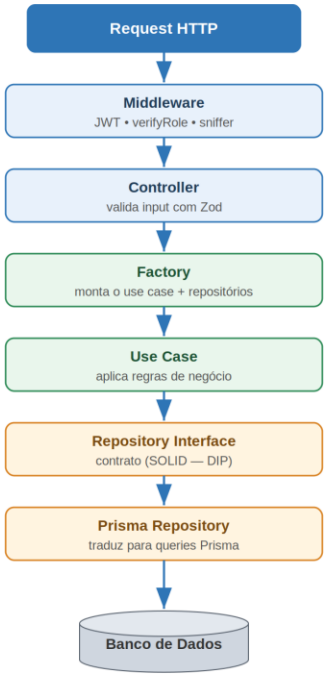

Figura 2 — Caminho completo de uma requisição, do middleware ao banco de dados.

4. Pontos de Entrada da Aplicação

src/server.ts

O que é: ponto de entrada da aplicação.

Por que existe: responsável por iniciar o servidor Fastify, conectar ao MongoDB e definir host/porta. Separa a configuração do servidor (app.ts) da sua execução.

src/app.ts

O que é: configuração central do Fastify.

Por que existe: registra todos os plugins (CORS, JWT, multipart, static files), hooks globais (sniffer), rotas e o error handler global. É o esqueleto HTTP sem iniciar o servidor — o que permite importar o app em testes sem levantar a porta.

5. As Camadas em Detalhe

5.1 Extensões de Tipos — src/@types/

Arquivo	O que faz
`fastify.d.ts`	Estende FastifyRequest para incluir campos como logEntryUser, logEntryId, logEntryError e multipartData, permitindo que middlewares anexem dados ao request de forma tipada.
`fastify-jwt.d.ts`	Define o formato do payload JWT (sub, role, type), garantindo tipagem ao acessar request.user.
`serializes-error.d.ts`	Define a interface para serialização de erros.

Por que existem: o TypeScript precisa saber que request.logEntryUser existe. Sem esses arquivos, todo acesso a campos customizados daria erro de compilação.

5.2 Configuração de Ambiente — src/env/index.ts

Valida e exporta as variáveis de ambiente usando Zod. Garante que a aplicação não inicie com variáveis faltando ou inválidas. Em vez de `process.env.JWT_SECRET` (que pode ser undefined), usa-se `env.JWT_SECRET` (sempre string). A validação acontece no boot — se algo estiver errado, a aplicação falha imediatamente com uma mensagem clara.

5.3 Tratamento de Erros — src/error/

Arquivo	O que faz
`app-error.ts`	Define a classe AppError (erros de negócio com statusCode e mensagem segura) e sendError(), o error handler global do Fastify.
`handle-prisma-error.ts`	Mapeia códigos do Prisma (P2002 = duplicado → 409, P2025 = não encontrado → 404, etc.) para respostas HTTP seguras.
`normalize.ts`	Normaliza qualquer tipo de erro (Error, string, objeto) para um formato consistente de logging.

Por que existem: centralizam o tratamento de erros. Controllers e use cases apenas lançam erros (throw new AppError(...)) — nunca decidem como formatar a resposta. Isso evita try/catch duplicado e garante que erros internos (SQL, stack traces) nunca vazem para o cliente.

5.4 Integrações Externas — src/lib/

Arquivo	O que faz
`prisma.ts`	Cria e exporta o singleton do PrismaClient com adapter MariaDB. Ativa logging de queries em dev.
`mongoose.ts`	Exporta connectMongo() para conexão com o MongoDB.
`ati-crypto.ts`	Implementa a descriptografia AES-256-CBC do id_ati recebido do sistema ATI.

Por que existem: isolam a configuração de bibliotecas externas. Se o banco mudar de MariaDB para PostgreSQL, só estes arquivos mudam — nenhum controller ou use case precisa ser alterado.

5.5 Schemas do Mongoose — src/schemas/

`requests.ts` define o schema da collection requests no MongoDB, usada pelo sniffer para logging. O MongoDB é usado exclusivamente para auditoria de requisições.

5.6 Middlewares — src/http/middlewares/

Arquivo	O que faz
`jwt.ts`	Verifica o token JWT, extrai o payload, busca o usuário/operador via ControlAccessUseCase e popula request.logEntryUser.
`verifyRole.ts`	Verifica se o role do JWT está na lista permitida. Retorna 403 caso contrário.
`sniffer.ts`	Hook onRequest: registra a requisição no MongoDB. Hook onSend: atualiza com resposta, duração e erros.

Por que existem: implementam cross-cutting concerns — autenticação, autorização e logging que atravessam todas as rotas sem poluir os controllers.

5.7 Controllers — src/http/controllers/

Organizados por domínio (auth, users, categories, medias, requests, test). Cada domínio tem um routes.ts e controllers individuais por ação.

O que cada controller faz:

1. Valida o input (body, params, query) com schemas Zod
2. Chama a factory para obter o use case
3. Executa o método do use case
4. Retorna a resposta HTTP

Por que um arquivo por ação? evita arquivos grandes e monolíticos. Cada ação tem seu próprio schema de validação e pode ser encontrada ou modificada sem afetar as demais.

Domínios existentes:

Domínio	Arquivos	Propósito
`auth/`	user-ati.ts, me.ts, routes.ts	Autenticação via ATI (criptografado), endpoint /auth/me
`users/`	CRUD (5 + routes)	Gerenciamento de usuários (apenas operadores)
`categories/`	CRUD (5 + routes)	Categorias de mídia
`medias/`	CRUD (5 + routes)	Mídias (vídeos e PDFs)
`requests/`	list, get, routes	Consulta de logs de requisição (MongoDB)
`test/`	routes.ts	Endpoints de teste (apenas dev/homologação)

5.8 Repositório — src/repository/

Interfaces (contratos)

Arquivo	O que define
`user-repository.ts`	findById, findByLogin, findByIdSar, findMany, create, update, softDelete. Exporta CreateUserData e UpdateUserData.
`operator-repository.ts`	findById, findByIdAti, create, update.
`media-category-repository.ts`	findById, findMany, create, update, softDelete.
`media-repository.ts`	findById, findMany (com filtros), create, update, softDelete, countByCategory.

Por que existem: são o coração do Dependency Inversion Principle (SOLID). Os use cases dependem dessas interfaces, não da implementação concreta do Prisma. Isso permite trocar o ORM sem alterar regras de negócio e criar implementações in-memory para testes.

Implementações Prisma — prisma/

Arquivo	O que faz
`prisma-user-repository.ts`	Implementa UserRepository traduzindo cada método em queries Prisma.
`prisma-operator-repository.ts`	Implementa OperatorRepository com Prisma.
`prisma-media-category-repository.ts`	Implementa MediaCategoryRepository com Prisma.
`prisma-media-repository.ts`	Implementa MediaRepository. Usa include: { media_category: true } na leitura.

Por que existem: encapsulam todo o código específico do Prisma/banco. Nenhuma outra camada sabe que o banco é MariaDB. Filtros como deleted_at: null (soft delete) ficam aqui, não nos controllers.

5.9 Casos de Uso — src/use-cases/

Arquivo	O que faz
`control-user.ts`	Regras para usuários: verificar login duplicado antes de criar, validar existência antes de atualizar/deletar.
`control-access.ts`	Autenticação: encontrar ou criar usuário/operador a partir do id_ati, buscar por ID para validação de JWT.
`control-media-category.ts`	Regras para categorias: impedir exclusão de categoria com mídias vinculadas.
`control-media.ts`	Regras para mídias: validar existência antes de atualizar/deletar.

Por que existem: é onde mora a lógica que não é HTTP e não é banco de dados. Exemplos: “não pode criar usuário com login duplicado”, “não pode deletar categoria com mídias”, “se o usuário não existe pelo id_ati, crie; se existe, atualize”.

Padrão de construção — constructor injection:

```
class ControlMediaCategoryUseCase {
    constructor(
        private mediaCategoryRepository: MediaCategoryRepository,
        private mediaRepository: MediaRepository,
    ) { }
}
```

Fábricas — factories/

Arquivo	O que faz
`make-control-user.ts`	Instancia PrismaUserRepository e injeta no ControlUserUseCase.
`make-control-access.ts`	Instancia PrismaUserRepository + PrismaOperatorRepository no ControlAccessUseCase.
`make-control-media-category.ts`	Instancia os repos de MediaCategory + Media no use case.
`make-control-media.ts`	Instancia PrismaMediaRepository no ControlMediaUseCase.

Por que existem: são o ponto de composição — onde as dependências concretas são decididas. Os controllers chamam makeControlUserUseCase() e recebem um use case pronto, substituindo um container de DI (tsyringe, inversify) por algo mais simples e explícito.

6. Anatomia de um Domínio CRUD

Todos os domínios seguem exatamente o mesmo padrão. A figura abaixo mostra os arquivos envolvidos em um CRUD completo e como eles se conectam entre as camadas.

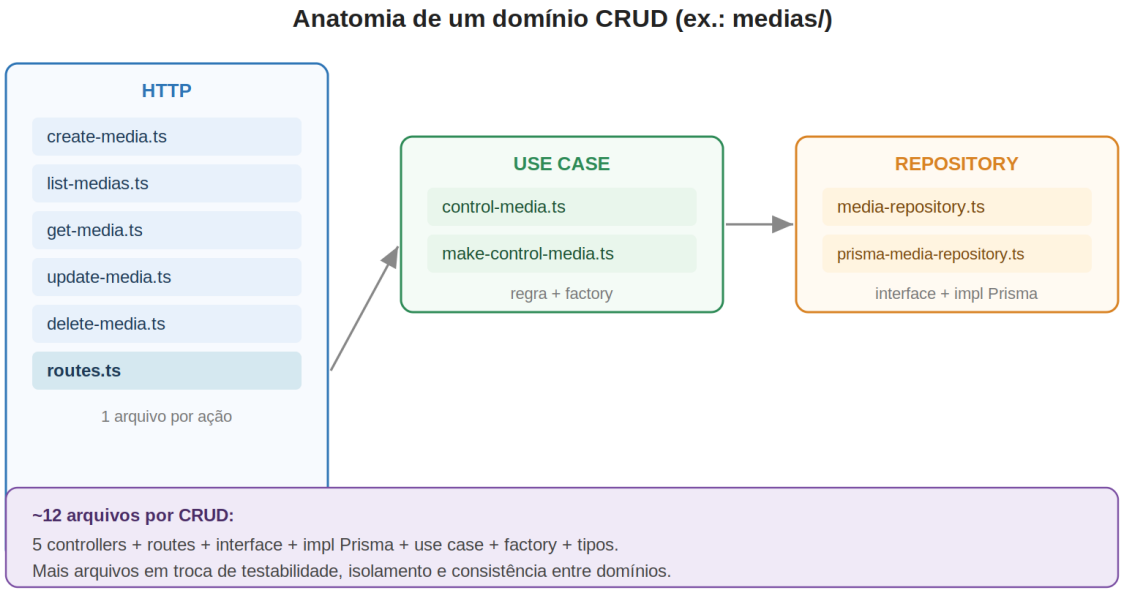

Figura 3 — Os arquivos de um domínio CRUD e suas dependências entre camadas.

7. Por Que Esta Arquitetura?

7.1 Vantagens

Vantagem	Explicação
Testabilidade	Use cases testáveis com repositórios in-memory, sem banco real — basta implementar as mesmas interfaces.
Separação de responsabilidades	Cada arquivo tem uma única razão para mudar. Mudança de schema afeta só o repositório; nova regra afeta só o use case.
Troca de tecnologia	Prisma → TypeORM, MariaDB → PostgreSQL, Fastify → Express: cada troca fica isolada em sua camada.
Facilidade de localização	Criação no banco? prisma-media-repository.ts. Regra de exclusão? control-media-category.ts. Rota POST? create-media.ts.
Consistência entre domínios	Todos os domínios seguem o mesmo padrão. Quem entende um, entende todos.
Erros centralizados	O handler global captura AppError, ZodError e erros Prisma automaticamente.
Segurança	Erros internos (SQL, stack traces) nunca chegam ao cliente; sendError() retorna mensagens seguras.
Soft delete uniforme	A lógica de deleted_at fica nos repositórios; nenhuma outra camada precisa lembrar de filtrar deletados.

7.2 Desvantagens

Desvantagem	Explicação
Mais arquivos	Um CRUD simples exige ~12 arquivos. Para protótipos pode parecer excesso de engenharia.
Indireção	Entender o fluxo completo exige navegar por 4-5 arquivos, aumentando a curva inicial.
Boilerplate repetitivo	Muitos use cases fazem “verificar se existe → chamar repositório”. Em CRUDs simples a camada pode parecer desnecessária.
Sem container de DI	As factories são manuais; com muitos repositórios ficam verbosas.
Performance marginal	Cada requisição instancia novos objetos via factory. Impacto negligível em Node.js, mas singleton seria mais eficiente.
Duplicação de tipos	Tipos existem nas interfaces dos repositórios e nos schemas Zod; mudar um campo exige atualizar ambos.

8. Quando Esta Arquitetura Faz Sentido

Ideal para:

- APIs com regras de negócio não-triviais
- Projetos com múltiplos desenvolvedores
- Sistemas que precisam de testes automatizados
- Projetos de longa duração que vão crescer

Pode ser excessiva para:

- Protótipos e MVPs descartáveis
- APIs puramente CRUD sem regras de negócio
- Projetos de um único desenvolvedor com prazo muito curto

9. Referência Rápida — Onde Fica Cada Coisa

Preciso...	Vou em...
Adicionar um novo endpoint	`src/http/controllers/{dominio}/ + routes.ts`
Mudar uma regra de negócio	`src/use-cases/control-*.ts`
Mudar como os dados são salvos	`src/repository/prisma/prisma-*-repository.ts`
Adicionar um novo campo ao banco	`schema.prisma + interface + impl Prisma`
Adicionar validação no input	`Schema Zod no controller`
Adicionar um novo domínio	`interface → impl → use case → factory → controllers → routes → app.ts`
Adicionar middleware global	`src/http/middlewares/ + app.ts`
Mudar tratamento de erros	`src/error/`
Mudar config do Fastify/plugins	`src/app.ts`
Mudar variáveis de ambiente	`src/env/index.ts + .env`