Estrutura de Pastas e Componentes da Arquitetura MVC

A solução foi estruturada no padrão de arquitetura Model-View-Controller (MVC) adaptado para uma API REST, utilizando Node.js e TypeScript. Esta organização visa garantir a Separação de Preocupações, isolando a interface (Controller), a lógica de negócio (Service) e o acesso a dados (Repository), o que facilita a manutenibilidade, a escalabilidade e os testes unitários.

Estrutura de Diretórios

src/

├── controllers/ # Lida com requisições HTTP

├── models/ # Define o contrato de dados (Entidade)

├── services/ # Contém a lógica de negócio

├── repositories/ # Abstrai o acesso à persistência

├── routes/ # Mapeamento de URI para Controller

├── config/ # Configurações do Express

└── app.ts # Inicialização do servidor

Para garantir que nossa API seja robusta, escalável e fácil de manter, adotamos a arquitetura em camadas, seguindo o princípio da Separação de Preocupações. Basicamente, isolamos a responsabilidade de cada componente, o que nos permite testar e evoluir cada parte do sistema de forma independente.

Vamos detalhar a função de cada diretório, usando o nosso recurso Produto como exemplo prático:

Começando pelo Domínio, que está em models/. É aqui que definimos o Contrato de Dados, nossa Entidade (Product.ts). O papel dela é fundamental: estabelecer a estrutura exata dos dados (como name, price, stock) e garantir a tipagem rigorosa do que entra e sai de todas as outras camadas, mantendo a consistência em todo o sistema.

Abaixo do Domínio, temos a camada de Persistência, localizada em repositories/. Esta é a implementação do Padrão Repository, cuja única responsabilidade é isolar a lógica de acesso a dados. O ProductRepository não se preocupa com regras de negócio; ele apenas sabe como executar o CRUD (Create, Read, Update, Delete) diretamente na nossa fonte de dados (atualmente, uma simulação em memória, mas poderia ser um banco de dados SQL ou NoSQL).

Acima da persistência e no centro do sistema, encontramos a camada de Lógica de Negócio em services/. O ProductService é onde as regras de domínio realmente acontecem. Ele recebe requisições do Controller, aplica todas as validações necessárias (por exemplo, checar se o price é positivo ou se há estoque disponível para uma operação) e coordena a comunicação com o Repository. Ele garante que a operação de persistência só ocorra se todas as regras de negócio forem satisfeitas.

No topo da pilha, servindo como a Interface de Apresentação, temos os controllers/. O ProductController é o único componente que tem contato direto com o protocolo HTTP. Ele recebe a requisição (req), extrai os dados do corpo ou da query, e os repassa ao Service. Ele nunca deve conter lógica de negócio; seu trabalho é apenas orquestrar, chamar o método apropriado do ProductService e, por fim, formatar a resposta (res) em JSON, devolvendo-a ao cliente com o Status HTTP correto.

Por fim, os diretórios routes/ e config/ lidam com a infraestrutura: routes/ mapeia cada URI (/products, /products/:id) para a função exata dentro do Controller, enquanto config/ centraliza a inicialização do Express e a configuração de middlewares globais, mantendo o arquivo principal (app.ts) limpo e focado apenas no bootstrap da aplicação.