

Atividade Prática

Este documento apresenta um guia para a implementação de uma API RESTful em C# utilizando ASP.NET Core, com foco na arquitetura em camadas e integração com Docker. A atividade propõe o desenvolvimento de um sistema com persistência em PostgreSQL, seguindo boas práticas de desenvolvimento e arquitetura de software.

Modelo de Negócio

O modelo de negócio da aplicação é composto por duas entidades principais que se relacionam entre si, formando a base para nosso domínio. Estas entidades serão mapeadas para tabelas no banco de dados PostgreSQL e servirão como estrutura fundamental para as operações da API.

Entidade Category

- Id (int, chave primária)
- Name (string, obrigatório)
- Description (string, opcional)

Entidade Product

- Id (int, chave primária)
- Name (string, obrigatório)
- Price (decimal, obrigatório)
- CategoryId (int, chave estrangeira)
- Category (propriedade de navegação)

O relacionamento entre estas entidades é do tipo um-para-muitos (1:N), onde uma categoria pode ter vários produtos associados a ela, enquanto um produto pertence a apenas uma categoria. Este relacionamento é fundamental para a integridade referencial do banco de dados e para a implementação correta das regras de negócio.

Na implementação com Entity Framework Core, utilizaremos a propriedade de navegação **Category** na entidade Product para facilitar o acesso aos dados relacionados, permitindo consultas mais expressivas e eficientes. A chave estrangeira **CategoryId** estabelecerá a relação formal entre as tabelas no banco de dados.

É importante considerar a validação dos campos obrigatórios tanto na camada de domínio quanto na persistência, garantindo que os dados inseridos no sistema estejam sempre consistentes com as regras estabelecidas.

Funcionalidades da API

Nossa API RESTful implementará um conjunto completo de endpoints para gerenciar as entidades Category e Product, seguindo os princípios de design de APIs e as melhores práticas do padrão REST. Cada endpoint será mapeado para um método específico em controladores [ASP.NET](#) Core.

Endpoints para Category

- GET /categories — Lista todas as categorias cadastradas
- GET /categories/{id} — Obtém uma categoria específica pelo ID
- POST /categories — Cria uma nova categoria
- PUT /categories/{id} — Atualiza uma categoria existente
- DELETE /categories/{id} — Remove uma categoria
- GET /categories/{id}/products — Lista todos os produtos de uma categoria

Cada um desses endpoints será implementado em controladores específicos, com os métodos HTTP apropriados. Os controladores serão responsáveis apenas pela camada de apresentação da API, delegando a lógica de negócio para a camada de serviço.

Para garantir a qualidade da API, implementaremos:

- Validação de entrada de dados
- Tratamento adequado de erros e exceções
- Códigos HTTP de status corretos para cada resposta
- Paginação para endpoints de listagem que podem retornar muitos registros
- Documentação via Swagger/OpenAPI

A implementação dessas funcionalidades seguirá o padrão RESTful, proporcionando uma interface consistente e intuitiva para os consumidores da API.

Arquitetura Esperada

A arquitetura do projeto segue o padrão de camadas, uma abordagem que promove a separação de responsabilidades e facilita a manutenção e evolução do sistema. Ela será estruturada em, no mínimo, três projetos distintos, cada um com responsabilidades bem definidas.

Projeto	Conteúdo Esperado
MonolitoBackend.Core	Entidades, interfaces de serviço, regras de domínio
MonolitoBackend.Infrastructure	DbContext, repositórios, implementações
MonolitoBackend.Api	Controladores, injeção de dependência, endpoints REST

Endpoints para Product

- GET /products — Lista todos os produtos
- GET /products/{id} — Obtém um produto específico pelo ID
- GET /products/by-category/{categoryId} — Lista produtos por categoria
- POST /products — Cria um novo produto
- PUT /products/{id} — Atualiza um produto existente
- DELETE /products/{id} — Remove um produto

A organização em camadas proporciona diversos benefícios para o projeto:



Separação de Responsabilidades

Cada camada tem uma função específica e bem definida, facilitando a compreensão e manutenção do código.



Independência de Implementação

A camada de domínio (Core) não depende de detalhes de implementação, permitindo que frameworks e tecnologias sejam substituídos sem afetar a lógica de negócio.



Reusabilidade

Componentes bem encapsulados podem ser reutilizados em diferentes partes do sistema ou mesmo em outros projetos.



Testabilidade

A separação clara de responsabilidades facilita a escrita de testes unitários e de integração mais eficazes.

Detalhamento das Camadas

1. **MonolitoBackend.Core:** Contém as entidades de domínio (Category, Product), interfaces de serviços e repositórios, e toda a lógica de negócio. Esta camada não deve ter dependências externas, especialmente de frameworks.
2. **MonolitoBackend.Infrastructure:** Implementa as interfaces definidas na camada Core, como repositórios e serviços. Aqui ficam as classes relacionadas ao Entity Framework Core, incluindo o DbContext, migrações e configurações de mapeamento objeto-relacional.
3. **MonolitoBackend.Api:** Expõe os endpoints REST, implementa os controladores e configura a injeção de dependência. Esta camada é responsável pela comunicação com o mundo externo, tratando requisições HTTP, validação de entrada e formatação de respostas.

Essa arquitetura em camadas é um exemplo do padrão Clean Architecture, que visa criar sistemas mais independentes de frameworks, testáveis e com clara separação de responsabilidades.

Requisitos com Docker

O Docker é uma ferramenta essencial para criar ambientes isolados e reproduzíveis, facilitando o desenvolvimento, teste e implantação de aplicações. Neste projeto, utilizaremos o Docker para gerenciar o banco de dados PostgreSQL e, opcionalmente, o próprio aplicativo [ASP.NET](#) Core.

Estrutura do docker-compose.yml

```
version: '3.8'

services:
  postgres:
    image: postgres:latest
    container_name: api-postgres
    environment:
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
      - POSTGRES_DB=${DB_NAME}
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql/data
    restart: always
    networks:
      - app-network

volumes:
  postgres-data:
    name: postgres-api-data

networks:
  app-network:
    driver: bridge
```

O arquivo docker-compose.yml define a configuração necessária para executar o banco de dados PostgreSQL em um container Docker. Os componentes principais dessa configuração são:

- **Container PostgreSQL:** Utiliza a imagem oficial mais recente do PostgreSQL.
- **Variáveis de Ambiente:** Configurações do banco como usuário, senha e nome do banco de dados são definidas através de variáveis de ambiente, aumentando a segurança e flexibilidade.
- **Volume Nomeado:** Os dados do PostgreSQL são armazenados em um volume persistente, garantindo que os dados não sejam perdidos quando o container for reiniciado.
- **Mapeamento de Portas:** A porta 5432 do container é mapeada para a mesma porta no host, permitindo a conexão com o banco de dados.
- **Rede:** Define uma rede bridge para comunicação entre containers, caso o aplicativo também seja containerizado.

Conectando o Projeto .NET ao PostgreSQL no Container

Para que o projeto .NET se conecte ao PostgreSQL no container, é necessário configurar corretamente a string de conexão no arquivo appsettings.json ou através de variáveis de ambiente:

```
"ConnectionStrings": {
  "DefaultConnection":
    "Host=localhost;Port=5432;Database=${DB_NAME};Username=${DB_USER};Password=${DB_PASSWORD};"
}
```

Opcional: Containerizando o Projeto .NET

Para containerizar o próprio projeto .NET, podemos adicionar um serviço adicional no docker-compose.yml:

```
api:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: api-dotnet
  ports:
    - "5000:80"
  depends_on:
    - postgres
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    -
  ConnectionStrings__DefaultConnection=Host=postgres;Port=5432;Database=${DB_NAME};Username=${DB_USER};Password=${DB_PASSWORD};
  networks:
    - app-network
```

Neste caso, será necessário também criar um Dockerfile na raiz do projeto para construir a imagem da aplicação .NET.

Critérios de Avaliação

A avaliação do projeto será baseada em diversos critérios que englobam desde a funcionalidade da API até a organização do código e a implementação das tecnologias requeridas. Compreender esses critérios é fundamental para garantir que todos os aspectos importantes sejam devidamente considerados durante o desenvolvimento.

Critério	Pontos
Funcionalidade dos endpoints	30
Separação em camadas e boas práticas	20
Utilização correta do Docker e docker-compose	25
Banco de dados funcionando e migrado corretamente	15
Organização do repositório e README.md	10

Detalhamento dos Critérios



Funcionalidade dos endpoints (30 pontos)

Todos os endpoints definidos devem funcionar corretamente, retornando os dados esperados e utilizando os métodos HTTP apropriados. A API deve implementar validações adequadas, tratar erros de forma consistente e retornar códigos de status HTTP corretos.

Separação em camadas e boas práticas (20 pontos)

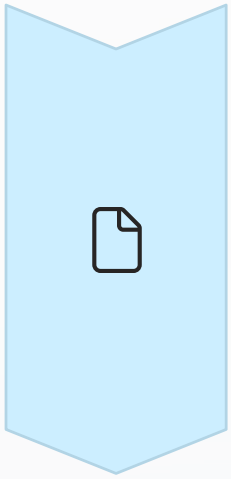
O código deve seguir a arquitetura em camadas especificada, com clara separação de responsabilidades. Padrões de projeto devem ser aplicados corretamente, como Repository, Dependency Injection e Service Layer. O código deve ser limpo, legível e seguir convenções de nomenclatura.

Utilização correta do Docker e docker-compose (25 pontos)

Os arquivos docker-compose.yml e Dockerfile (se aplicável) devem estar configurados corretamente. Os containers devem iniciar sem erros, e a comunicação entre eles deve funcionar adequadamente. As variáveis de ambiente devem ser utilizadas para configuração.

Banco de dados funcionando e migrado corretamente (15 pontos)

O banco de dados PostgreSQL deve ser inicializado corretamente no container. As migrações do Entity Framework Core devem criar o esquema do banco de dados com as tabelas, colunas e relacionamentos conforme o modelo de domínio. A persistência e recuperação de dados devem funcionar sem erros.



Organização do repositório e README.md (10 pontos)

O repositório deve ser organizado de forma clara, com estrutura de pastas lógica. O README.md deve conter instruções detalhadas sobre como executar o projeto com Docker, descrição dos endpoints da API com exemplos de uso, e possivelmente um diagrama da arquitetura.

Os critérios acima fornecem um norte claro para o desenvolvimento do projeto, destacando as áreas que serão consideradas mais importantes na avaliação. É recomendado revisar periodicamente esses critérios durante o desenvolvimento para garantir que todos os aspectos estejam sendo adequadamente endereçados.

Entrega

A entrega do projeto deve seguir algumas diretrizes específicas para garantir que o avaliador consiga compreender, executar e avaliar corretamente o trabalho desenvolvido. A documentação clara e objetiva é tão importante quanto o código em si, pois facilita a verificação das funcionalidades implementadas.

Publicação no GitHub

Todo o código-fonte do projeto deve ser publicado em um repositório no GitHub. Essa é uma prática comum no mercado de desenvolvimento e permite que o avaliador tenha acesso a todo o histórico de desenvolvimento através dos commits. Recomenda-se criar um repositório privado inicialmente e torná-lo público apenas no momento da entrega, ou conforme orientação específica do avaliador.

Elaboração do [README.md](#)

O arquivo [README.md](#) é a porta de entrada do seu projeto e deve conter todas as informações necessárias para que qualquer pessoa possa entender, configurar e executar a aplicação. Um bom [README.md](#) para este projeto deve incluir:



Instruções de Execução com Docker

Detalhe passo a passo como executar o projeto utilizando Docker. Inclua os comandos necessários para iniciar os containers, aplicar migrações e verificar se tudo está funcionando corretamente. Mencione também os requisitos de software (versões do Docker e Docker Compose) e as variáveis de ambiente que precisam ser configuradas.



Endpoints da API e Exemplos de Uso

Liste todos os endpoints disponíveis, agrupados por entidade, com descrição clara do que cada um faz. Para cada endpoint, forneça exemplos de requisições (cURL, Postman ou similar) e respostas esperadas. Inclua exemplos de casos de sucesso e também de erros comuns e como a API os trata.



Diagrama da Arquitetura (opcional)

Um diagrama visual da arquitetura pode facilitar muito a compreensão do projeto. Mesmo sendo opcional, é altamente recomendado incluir um diagrama simples mostrando como as diferentes camadas e componentes se relacionam. Ferramentas como Draw.io, Lucidchart ou mesmo o PlantUML podem ser utilizadas para criar estes diagramas.

Dicas para uma boa entrega

- **Clareza e objetividade:** Seja claro e objetivo na documentação. Evite textos longos e redundantes.
- **Formatação adequada:** Utilize corretamente a sintaxe do Markdown para criar uma documentação bem formatada e legível.
- **Exemplos práticos:** Sempre que possível, inclua exemplos práticos de uso da API.
- **Screenshots:** Adicione screenshots da aplicação em funcionamento, especialmente para demonstrar a execução com Docker.
- **Troubleshooting:** Inclua uma seção de solução de problemas comuns que podem ocorrer durante a execução.

Uma documentação bem elaborada não apenas facilita a avaliação do projeto, mas também demonstra profissionalismo e atenção aos detalhes, características altamente valorizadas no mercado de desenvolvimento de software.

Variações de Modelo de Negócio

O projeto permite a implementação de diferentes modelos de negócio além do padrão Categorias e Produtos. Essas variações possibilitam que você escolha um domínio que seja mais interessante ou familiar, mantendo a mesma estrutura arquitetural e tecnológica. A seguir, apresentamos as sete opções disponíveis.

Cada modelo alternativo segue o mesmo padrão: entidades principais relacionadas entre si, com endpoints REST para operações CRUD e consultas específicas. A escolha de uma dessas variações não altera os requisitos técnicos nem os critérios de avaliação do projeto.

Biblioteca Digital

Entidades:	Endpoints:
<ul style="list-style-type: none">• Book (Livro): Id, Title, Author, ISBN, PublishedYear• Genre (Gênero): Id, Name, Description	<ul style="list-style-type: none">• GET /genres - Listar todos os gêneros• GET /genres/{id} - Obter gênero por ID• POST /genres - Criar novo gênero• PUT /genres/{id} - Atualizar gênero• DELETE /genres/{id} - Remover gênero• GET /genres/{id}/books - Listar livros do gênero• GET /books - Listar todos os livros• GET /books/{id} - Obter livro por ID• GET /books/by-genre/{genreId} - Listar livros por gênero• POST /books - Criar novo livro• PUT /books/{id} - Atualizar livro• DELETE /books/{id} - Remover livro

Este modelo é ideal para aplicações como sistemas de bibliotecas, catálogos literários ou plataformas de recomendação de livros. A API permite gerenciar um acervo de livros classificados por gêneros, facilitando a organização e busca de títulos.

Implementação do Modelo

Para implementar o modelo de Biblioteca Digital, você precisará criar as classes de entidade Book e Genre, configurar o relacionamento entre elas no DbContext e implementar os repositórios e serviços correspondentes. A lógica de negócio pode incluir regras como validação de ISBN, verificação de duplicidade de livros e integridade referencial entre livros e gêneros.

Este modelo apresenta uma complexidade adequada para demonstrar o uso da arquitetura em camadas e o relacionamento entre entidades, sendo bastante didático para o aprendizado de desenvolvimento de APIs RESTful com ASP.NET Core.

Sistema de Reservas de Salas

O Sistema de Reservas de Salas representa a segunda variação possível para o modelo de negócio do projeto. Este domínio é particularmente útil para aplicações corporativas, educacionais ou de coworking, onde há necessidade de gerenciar a disponibilidade e ocupação de espaços físicos.

Entidades

Room (Sala):

- Id (int, chave primária)
- Name (string, obrigatório)
- Capacity (int, obrigatório)
- HasProjector (bool)

Reservation (Reserva):

- Id (int, chave primária)
- RoomId (int, chave estrangeira)
- ReservedBy (string, obrigatório)
- StartTime (DateTime, obrigatório)
- EndTime (DateTime, obrigatório)

Relacionamento: Uma sala pode ter várias reservas (1:N)

Endpoints da API

Para Room (Sala):

- GET /rooms - Listar todas as salas
- GET /rooms/{id} - Obter sala por ID
- POST /rooms - Criar nova sala
- PUT /rooms/{id} - Atualizar sala
- DELETE /rooms/{id} - Remover sala
- GET /rooms/{id}/reservations - Listar reservas da sala

Para Reservation (Reserva):

- GET /reservations - Listar todas as reservas
- GET /reservations/{id} - Obter reserva por ID
- GET /reservations/by-room/{roomId} - Listar reservas por sala
- POST /reservations - Criar nova reserva
- PUT /reservations/{id} - Atualizar reserva
- DELETE /reservations/{id} - Remover reserva

Regras de Negócio Específicas

Este modelo introduz algumas regras de negócio específicas que podem ser implementadas para enriquecer a aplicação:

- **Validação de conflitos:** Verificar se uma nova reserva não conflita com reservas existentes para a mesma sala (sobreposição de horários).
- **Validação de horários:** Garantir que o horário de término seja posterior ao horário de início.
- **Capacidade da sala:** Implementar lógica para verificar se a capacidade da sala é adequada para o evento (poderia ser adicionado um campo de número de participantes na reserva).
- **Recursos especiais:** Filtrar salas por recursos disponíveis, como projetor (HasProjector).

Implementação Técnica

Na implementação do Sistema de Reservas de Salas, o foco da camada de serviços estará na validação das regras de negócio mencionadas acima. Os repositórios precisarão suportar consultas específicas, como:

- Buscar salas disponíveis em um determinado período
- Verificar reservas futuras para uma sala específica
- Listar reservas de um determinado usuário (ReservedBy)

Este modelo é ideal para demonstrar o uso de tipos de dados DateTime no Entity Framework Core, bem como validações de negócio mais complexas que envolvem comparações de intervalos temporais.

Gestão de Tarefas e Projetos

O modelo de Gestão de Tarefas e Projetos representa a terceira variação do projeto, sendo especialmente útil para aplicações de gerenciamento de trabalho e produtividade. Este domínio é relevante tanto para equipes quanto para uso pessoal, permitindo organizar trabalhos em projetos e acompanhar o progresso através de tarefas individuais.

Entidades

Project:

- Id (int, chave primária)
- Name (string, obrigatório)
- StartDate (DateTime, obrigatório)
- EndDate (DateTime, opcional)

Task:

- Id (int, chave primária)
- ProjectId (int, chave estrangeira)
- Title (string, obrigatório)
- Status (string/enum, obrigatório)
- DueDate (DateTime, opcional)

Relacionamento: Um projeto contém várias tarefas (1:N)

Endpoints da API

Para Project:




- GET /projects - Listar todos os projetos
- GET /projects/{id} - Obter projeto por ID
- POST /projects - Criar novo projeto
- PUT /projects/{id} - Atualizar projeto
- DELETE /projects/{id} - Remover projeto
- GET /projects/{id}/tasks - Listar tarefas do projeto

Para Task:

- GET /tasks - Listar todas as tarefas
- GET /tasks/{id} - Obter tarefa por ID
- GET /tasks/by-project/{projectId} - Listar tarefas por projeto
- POST /tasks - Criar nova tarefa
- PUT /tasks/{id} - Atualizar tarefa
- DELETE /tasks/{id} - Remover tarefa

Implementação do Status de Tarefas

Uma característica interessante deste modelo é a implementação do campo Status na entidade Task. Isso pode ser feito de várias maneiras:

	<h3>Utilizando Enum</h3> <p>Definir um enum em C# para representar os possíveis estados da tarefa (Ex: ToDo, InProgress, Done, Blocked). Esta abordagem proporciona forte tipagem e facilita a validação.</p>		<h3>Utilizando String com Validação</h3> <p>Armazenar o status como string e implementar validação para garantir que apenas valores válidos sejam aceitos. Esta abordagem é mais flexível para alterações futuras.</p>		<h3>Utilizando Tabela de Referência</h3> <p>Criar uma entidade StatusType separada e estabelecer um relacionamento com Task. Esta abordagem é ideal se os status forem dinâmicos ou tiverem propriedades adicionais.</p>
---	---	---	--	---	--

Funcionalidades Adicionais

Ao implementar este modelo, considere adicionar algumas funcionalidades que enriqueceriam a aplicação:

- **Filtro de tarefas:** Permitir filtrar tarefas por status, data de vencimento ou outros critérios.
- **Estatísticas de projeto:** Endpoint para obter métricas como percentual de conclusão, tarefas em atraso, etc.
- **Transições de status:** Implementar regras para validar transições de status (ex: uma tarefa só pode ir para "Done" se estiver em "InProgress").
- **Histórico de alterações:** Registrar mudanças de status e atualizações importantes nas tarefas.

Este modelo é particularmente interessante para demonstrar o uso de enums no Entity Framework Core, bem como a implementação de regras de negócio que envolvem transições de estado e validações temporais.

Controle de Estudantes e Matrículas

O modelo de Controle de Estudantes e Matrículas representa a quarta variação do projeto, sendo especialmente útil para sistemas educacionais, plataformas de ensino online e controle acadêmico. Este domínio introduz um relacionamento mais complexo: muitos-para-muitos (N:N) entre estudantes e cursos, implementado através da entidade de junção Enrollment.

Entidades

Student:

- Id (int, chave primária)
- Name (string, obrigatório)
- Email (string, obrigatório)
- EnrollmentDate (DateTime, obrigatório)

Course:

- Id (int, chave primária)
- Name (string, obrigatório)
- WorkloadHours (int, obrigatório)

Enrollment (Matrícula):

- StudentId (int, parte da chave primária composta)
- CourseId (int, parte da chave primária composta)
- EnrolledOn (DateTime, obrigatório)

Relacionamento: Muitos-para-muitos (N:N) entre estudantes e cursos

Endpoints da API

Para Student:

- GET /students - Listar todos os estudantes
- GET /students/{id} - Obter estudante por ID
- POST /students - Criar novo estudante
- PUT /students/{id} - Atualizar estudante
- DELETE /students/{id} - Remover estudante
- GET /students/{id}/courses - Listar cursos do aluno

Para Course:

- GET /courses - Listar todos os cursos
- GET /courses/{id} - Obter curso por ID
- POST /courses - Criar novo curso
- PUT /courses/{id} - Atualizar curso
- DELETE /courses/{id} - Remover curso
- GET /courses/{id}/students - Listar alunos do curso

Para Enrollment:

- POST /enrollments - Vincular estudante e curso
- DELETE /enrollments - Remover matrícula (via corpo JSON)

Implementação do Relacionamento N:N

Este modelo é particularmente valioso para demonstrar como implementar relacionamentos muitos-para-muitos no Entity Framework Core. Existem duas abordagens principais:



Tabela de Junção Explícita

Criar uma entidade Enrollment completa, com suas próprias propriedades além das chaves estrangeiras. Esta é a abordagem usada neste modelo, permitindo armazenar informações adicionais sobre a matrícula, como a data.



Tabela de Junção Implícita

Permitir que o EF Core crie automaticamente a tabela de junção, definindo apenas coleções de navegação nas entidades Student e Course. Esta abordagem é mais simples, mas limita a capacidade de armazenar dados adicionais sobre a relação.

Considerações Especiais

Ao implementar este modelo, é importante considerar alguns aspectos específicos:

- **Validação de matrículas:** Verificar se um estudante já está matriculado em um curso antes de criar uma nova matrícula.
- **Cascata de exclusão:** Decidir o comportamento quando um estudante ou curso é excluído (excluir todas as matrículas associadas ou impedir a exclusão).
- **Performance de consultas:** Optimizar consultas que envolvem múltiplos joins entre as três tabelas.
- **Paginação:** Implementar paginação eficiente para listas de estudantes por curso ou cursos por estudante, que podem crescer significativamente.

Este modelo é ideal para demonstrar técnicas mais avançadas de modelagem de dados com Entity Framework Core, além de fornecer um exemplo concreto de como trabalhar com relacionamentos muitos-para-muitos em APIs RESTful.

Sistema de Vendas Online

O modelo de Sistema de Vendas Online representa a quinta variação do projeto, sendo relevante para aplicações de e-commerce, gestão de pedidos e relacionamento com clientes. Este domínio aborda um cenário comercial comum, com clientes realizando múltiplos pedidos ao longo do tempo.

Entidades

Customer:

- Id (int, chave primária)
- Name (string, obrigatório)
- Email (string, obrigatório)
- Phone (string, opcional)

Order:

- Id (int, chave primária)
- CustomerId (int, chave estrangeira)
- OrderDate (DateTime, obrigatório)
- Total (decimal, obrigatório)

Relacionamento: Um cliente pode ter vários pedidos (1:N)

Endpoints da API

Para Customer:

- GET /customers - Listar todos os clientes
- GET /customers/{id} - Obter cliente por ID
- POST /customers - Criar novo cliente
- PUT /customers/{id} - Atualizar cliente
- DELETE /customers/{id} - Remover cliente
- GET /customers/{id}/orders - Listar pedidos do cliente

Para Order:

- GET /orders - Listar todos os pedidos
- GET /orders/{id} - Obter pedido por ID
- GET /orders/by-customer/{customerId} - Listar pedidos por cliente
- POST /orders - Criar novo pedido
- PUT /orders/{id} - Atualizar pedido
- DELETE /orders/{id} - Remover pedido

Expansão Possível do Modelo

Este modelo básico pode ser expandido para incluir funcionalidades mais complexas de um sistema de vendas completo:

Itens do Pedido Adicionar uma entidade OrderItem para representar os itens individuais em cada pedido, com campos como ProductId, Quantity e UnitPrice. Isso criaria um relacionamento 1:N entre Order e OrderItem.	Produtos Incluir uma entidade Product completa com Name, Description, Price e outras propriedades. Os OrderItems fariam referência a estes produtos, permitindo o controle de estoque e catálogo.	Status do Pedido Adicionar um campo Status na entidade Order para acompanhar o progresso (ex: "Criado", "Pago", "Enviado", "Entregue", "Cancelado"), com endpoints adicionais para atualizar o status.
---	---	--

Considerações para Implementação

Na implementação deste modelo, alguns aspectos merecem atenção especial:

- Validação de e-mail e telefone:** Implementar validações específicas para garantir a formatação correta destes campos no Customer.
- Cálculo automático do Total:** Na versão expandida, o campo Total em Order poderia ser calculado automaticamente com base nos itens do pedido.
- Consistência de dados:** Estabelecer regras para evitar a exclusão de clientes que possuem pedidos ativos.
- Auditoria:** Considerar a implementação de campos de auditoria (CreatedAt, UpdatedAt) para rastrear alterações nas entidades.

Este modelo é particularmente útil para demonstrar o uso de tipos de dados decimais para valores monetários, bem como a implementação de regras de negócio que envolvem cálculos e validações complexas. Além disso, oferece oportunidades para explorar relatórios e análises, como total de vendas por período, histórico de compras de clientes e métricas de desempenho.

Controle de Inventário

O modelo de Controle de Inventário representa a sexta variação do projeto, sendo especialmente relevante para aplicações de gestão de estoque, sistemas de almoxarifado e controle de suprimentos. Este domínio aborda um cenário comum em empresas de diversos segmentos, permitindo o gerenciamento eficiente de itens e seus fornecedores.

Entidades

Item:

- Id (int, chave primária)
- Name (string, obrigatório)
- SKU (string, obrigatório)
- Quantity (int, obrigatório)
- UnitPrice (decimal, obrigatório)
- SupplierId (int, chave estrangeira)

Supplier:

- Id (int, chave primária)
- Name (string, obrigatório)
- CNPJ (string, obrigatório)
- Phone (string, opcional)

Relacionamento: Um fornecedor fornece vários itens (1:N)

Endpoints da API

Para Supplier:

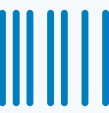
- GET /suppliers - Listar todos os fornecedores
- GET /suppliers/{id} - Obter fornecedor por ID
- POST /suppliers - Criar novo fornecedor
- PUT /suppliers/{id} - Atualizar fornecedor
- DELETE /suppliers/{id} - Remover fornecedor
- GET /suppliers/{id}/items - Listar itens fornecidos

Para Item:

- GET /items - Listar todos os itens
- GET /items/{id} - Obter item por ID
- GET /items/by-supplier/{supplierId} - Listar itens por fornecedor
- POST /items - Criar novo item
- PUT /items/{id} - Atualizar item
- DELETE /items/{id} - Remover item

Funcionalidades Específicas do Sistema de Inventário

Um sistema de controle de inventário pode ser enriquecido com diversas funcionalidades específicas:



Validação de SKU

Implementar validação para garantir que o SKU (Stock Keeping Unit) seja único e siga um formato específico, facilitando a identificação e rastreamento dos itens no estoque.



Movimentação de Estoque

Adicionar endpoints para registrar entradas e saídas de itens, ajustando automaticamente a quantidade disponível e mantendo um histórico de transações.



Alertas de Estoque

Implementar um sistema de alertas para notificar quando a quantidade de um item estiver abaixo de um limite mínimo definido, evitando quebras de estoque.



Valoração de Estoque

Criar um endpoint para calcular o valor total do estoque, multiplicando a quantidade pelo preço unitário de cada item, auxiliando no controle financeiro.

Validações e Regras de Negócio

Na implementação deste modelo, algumas validações e regras de negócio são especialmente importantes:

- **Validação de CNPJ:** Implementar um validador para garantir que o CNPJ dos fornecedores seja válido e esteja no formato correto.
- **Quantidade Não Negativa:** Garantir que a quantidade de itens nunca seja negativa, implementando validações antes de qualquer operação de redução de estoque.
- **Persistência de Fornecedores:** Estabelecer regras para impedir a exclusão de fornecedores que possuem itens ativos no estoque.
- **Histórico de Preços:** Considerar a implementação de um mecanismo para rastrear mudanças de preço ao longo do tempo, garantindo transparência nas alterações.

Este modelo é particularmente interessante para demonstrar a aplicação de validações de dados específicas do domínio brasileiro (como CNPJ) e o gerenciamento de recursos com restrições quantitativas. Além disso, oferece oportunidades para implementar lógicas de negócio relacionadas a cálculos financeiros e controle de estoque.

Sistema de Check-in de Eventos

O modelo de Sistema de Check-in de Eventos representa a sétima e última variação do projeto, sendo especialmente útil para aplicações de gestão de eventos, conferências e encontros. Este domínio aborda um cenário comum na organização de eventos de qualquer porte, permitindo o controle eficiente dos participantes e sua presença nos eventos.

Entidades

Event:

- Id (int, chave primária)
- Title (string, obrigatório)
- Location (string, obrigatório)
- Date (DateTime, obrigatório)

Participant:

- Id (int, chave primária)
- Name (string, obrigatório)
- Email (string, obrigatório)
- EventId (int, chave estrangeira)

Relacionamento: Um evento possui vários participantes (1:N)

Endpoints da API

Para Event:

- GET /events - Listar todos os eventos
- GET /events/{id} - Obter evento por ID
- POST /events - Criar novo evento
- PUT /events/{id} - Atualizar evento
- DELETE /events/{id} - Remover evento
- GET /events/{id}/participants - Listar participantes do evento

Para Participant:

- GET /participants - Listar todos os participantes
- GET /participants/{id} - Obter participante por ID
- GET /participants/by-event/{eventId} - Listar participantes por evento
- POST /participants - Criar novo participante
- PUT /participants/{id} - Atualizar participante
- DELETE /participants/{id} - Remover participante

Expansão do Modelo Base

O modelo básico apresentado pode ser expandido para incluir funcionalidades mais avançadas de gestão de eventos:

Check-in Efetivo

Adicionar um campo `CheckInTime` à entidade `Participant` para registrar quando o participante efetivamente chegou ao evento, com um endpoint específico para realizar o check-in.

Categorias de Participantes

Incluir um campo `Category` para classificar os participantes (ex: "VIP", "Palestrante", "Participante Regular"), permitindo tratamentos diferenciados.

Sessões do Evento

Criar uma entidade `Session` relacionada ao `Event`, para representar diferentes atividades ou palestras dentro do mesmo evento, com seu próprio controle de presença.

Funcionalidades Adicionais Potenciais



Considerações para Implementação

Na implementação deste modelo, alguns aspectos merecem atenção especial:

- **Validação de e-mail único:** Garantir que um mesmo e-mail não seja registrado múltiplas vezes para o mesmo evento.
- **Limites de capacidade:** Implementar verificação de capacidade máxima do evento antes de permitir novos registros.
- **Eventos passados:** Estabelecer regras para impedir inscrições em eventos que já ocorreram.
- **Notificações:** Considerar a implementação de um sistema de notificações por e-mail para confirmar inscrições ou lembrar sobre eventos próximos.

Este modelo é particularmente útil para demonstrar a implementação de funcionalidades relacionadas a datas e horários, bem como a gestão de relacionamentos entre entidades em um contexto de tempo real. Além disso, oferece oportunidades para explorar integrações com sistemas externos, como serviços de e-mail ou geração de documentos.