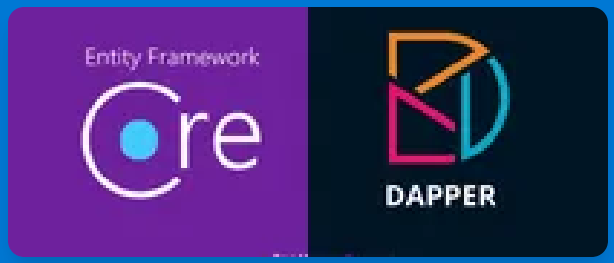
# Entity Framework Core

Banco de Dados: Modelagem, Persistência e Boas Práticas

## Agenda:

- Introdução a ORM e EF Core
- Migrations e Mapeamento de Entidades
- Operações Básicas do EF Core (CRUD)
- Relacionamentos (1:N)
- Padrão Repository (Introdução)




**Pré-requisitos:**

.NET SDK, Visual Studio/VS Code, SQL Server (ou SQLite), Postman/Insomnia

**Resultado esperado:**

API funcional com persistência via EF Core, evoluindo a cada aula

# Introdução ao Banco de Dados e EF Core

## O que é ORM

- Object-Relational Mapping: ponte entre objetos e tabelas
- Reduz código SQL repetitivo e aumenta produtividade
- Foco no domínio da aplicação ao invés de detalhes do banco

## DbContext

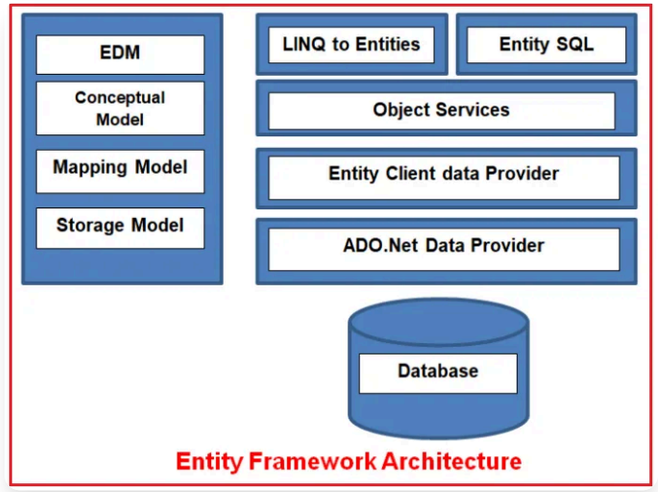
- Sessão com o banco de dados (unidade de trabalho)
- Gerencia ciclo de vida das entidades e tracking de mudanças
- Expõe DbSets para acessar coleções de entidades

## Instalação e Pacotes

- Microsoft.EntityFrameworkCore (pacote principal)
- Provider específico (SQL Server, PostgreSQL, SQLite...)
- Ferramentas para Migrations (dotnet ef / EF Core Tools)

## Configuração

- Connection string via appsettings.json ou secrets
- AddDbContext com injeção de dependência (Scoped)
- OnConfiguring / OnModelCreating para personalização




### Vantagens do EF Core:

- LINQ para consultas fortemente tipadas
- Migrations para gestão de schema
- Múltiplos providers de banco
- Performance otimizada
- Open source e integração com .NET

# Exercício 1 — Adicionar EF Core e configurar DbContext

**Objetivo:**

Adicionar Entity Framework Core ao projeto API e configurar o BankContext no Program.cs.

**Passos:**

1. **Adicionar pacotes NuGet:**
   - Microsoft.EntityFrameworkCore
   - Microsoft.EntityFrameworkCore.SqlServer (ou outro provider)
   - Microsoft.EntityFrameworkCore.Tools
2. **Criar connection string:**

   Adicionar em appsettings.json ou configurar via Secret Manager
3. **Criar a classe BankContext:**

   Herdar de DbContext e adicionar construtor com DbContextOptions
   Preparar propriedades DbSet para futuras entidades
4. **Registrar no Program.cs:**

   Usar AddDbContext e configurar o provider com a connection string
5. **Validar a configuração:**

   Criar endpoint de teste que resolve o DbContext

**Critérios de Aceite:**

- Aplicação inicia sem erros relacionados ao DbContext
- DbContext pode ser injetado em controladores
- Endpoint de teste retorna sucesso ao resolver o DbContext

# Migrations e Mapeamento de Entidades

## Code-First

- Classes C# como fonte primária do esquema de banco
- Foco no domínio primeiro, banco gerado a partir dele
- Versionamento do esquema integrado ao código

## Comandos de Migration

- Add-Migration [Nome] - Gera snapshot das mudanças
- Update-Database - Aplica migrations pendentes
- Script-Migration - Gera SQL sem executar

## Mapeamento Básico

- Convenções: Id/EntityNameId como chave primária
- Propriedades escalares (string, int, decimal) → colunas
- Personalização de tipos, tamanhos e precisão

## Data Annotations vs Fluent API

- Annotations: [Required], [MaxLength], [Column] - junto da classe
- Fluent API: OnModelCreating centralizado e mais poderoso
- Fluent API tem precedência sobre annotations


### Boas Práticas:

- Uma migration por feature/funcionalidade
- Use nomes descritivos (Add[Feature], Update[Feature])
- Sempre revise o código Up/Down gerado
- Evite dados sensíveis em seeds
- Teste migrations antes de compartilhar

# Operações Básicas do EF Core (CRUD)

## Create (Inserção)

- Add/AddAsync no DbSet para adicionar entidade
- AddRange/AddRangeAsync para múltiplas entidades
- SaveChanges/SaveChangesAsync para persistir no banco

## Read (Consulta)

- LINQ para consultas fortemente tipadas
- AsNoTracking para melhorar performance em consultas somente-leitura
- Skip/Take para paginação, Include para carregar dados relacionados

## Update (Atualização)

- Entidade rastreada: modifique propriedades e chame SaveChanges
- Update ou Entry().State = EntityState.Modified para entidades desanexadas
- Tratamento de concorrência: TimeStamp/RowVersion

## Delete (Remoção)

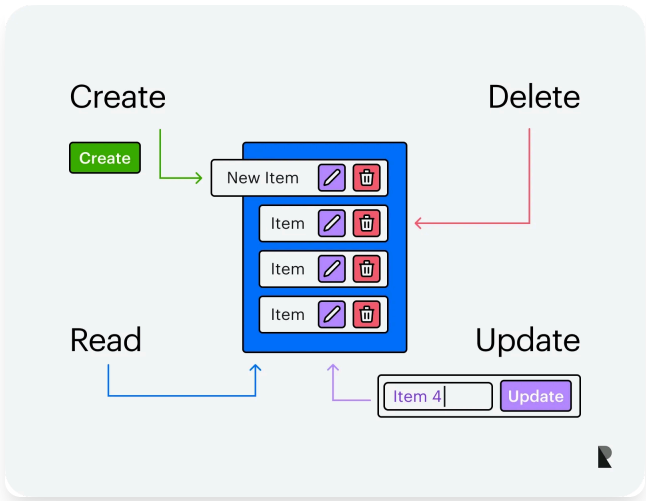
- Remove/RemoveAsync para entidade carregada
- RemoveRange para múltiplas entidades
- Configuração de delete cascade para entidades relacionadas

## Change Tracker

- Rastreia estados: Added, Modified, Unchanged, Deleted e Detached
- ChangeTracker.Entries() para acessar entidades rastreadas
- Impacto em performance: usar AsNoTracking quando apropriado

## Transações

- SaveChanges usa transação implícita por padrão
- BeginTransaction/UseTransaction para controle explícito
- Transações distribuídas com System.Transactions




**Erros comuns a evitar:**

- Compartilhar contexto entre threads
- Esquecer de usar await com métodos assíncronos
- Problema N+1 por não usar Include
- SaveChanges incompleto por exceção
- Consultas ineficientes sem filtros adequados

# Exercício 2 — Entidade Conta e primeira Migration

**Objetivo:**

Mapear a entidade Conta e criar o schema inicial do banco de dados.

**Passos:**

1. **Criar a classe Conta:**
   - Adicionar propriedades essenciais: Id, Numero, Saldo, DataCriacao, Status
   - Usar tipos apropriados para cada propriedade (decimal para Saldo, etc.)
2. **Definir anotações e validações:**
   - Usar Data Annotations como [Required], [MaxLength], [Column]
   - Ou configurar via Fluent API no método OnModelCreating do BankContext
3. **Adicionar DbSet ao BankContext:**
   - Criar propriedade DbSet<Conta> Contas { get; set; }
4. **Criar e aplicar Migration:**
   - Executar comando Add-Migration CriacaoTabelaConta
   - Aplicar ao banco com Update-Database
5. **Validar o resultado:**
   - Verificar a tabela gerada e suas colunas no banco de dados
   - Documentar a migration criada para referência futura

**Critérios de Aceite:**

- Tabela Conta criada corretamente no banco com todas as colunas
- Migration bem documentada e sem erros
- Constraints (como NOT NULL) corretamente aplicadas

# Exercício 3 — CRUD de Contas via DbContext

**Objetivo:**

Substituir a lista em memória nos Controllers por operações reais do DbContext.

**Passos:**

1. **Injetar o BankContext:**
   Adicionar construtor no ContasController que receba e armazene o BankContext

2. **Implementar endpoints CRUD:**
   - Criar (POST): Adicionar nova conta via Add/AddAsync
   - Listar (GET): Retornar todas as contas com filtros opcionais
   - Buscar por Id (GET/{id}): Encontrar conta específica
   - Atualizar (PUT/PATCH): Modificar conta existente
   - Remover (DELETE): Excluir conta por Id

3. **Utilizar métodos assíncronos:**
   - Usar SaveChangesAsync para operações de escrita
   - Aplicar AsNoTracking() em consultas somente-leitura para otimizar performance

4. **Implementar validações e HTTP status:**
   - 400 (Bad Request): Para entradas inválidas
   - 404 (Not Found): Quando conta não existe
   - 201 (Created): Ao criar com sucesso
   - 204 (No Content): Para operações sem retorno de conteúdo

5. **Adicionar recursos avançados:**
   - Paginação: Implementar com parâmetros page/size
   - Ordenação: Permitir ordenar por diferentes campos

**Critérios de Aceite:**

- CRUD completo funcional persistindo dados no banco
- Respostas HTTP adequadas para cada operação
- Paginação e filtros funcionando corretamente
- Testes bem-sucedidos via Postman/Insomnia

# Relacionamentos 1:N no EF Core

## Conceitos

- Chave primária (PK) e chave estrangeira (FK)
- Integridade referencial entre tabelas
- Propriedades de navegação e coleções

## Mapeamento 1:N

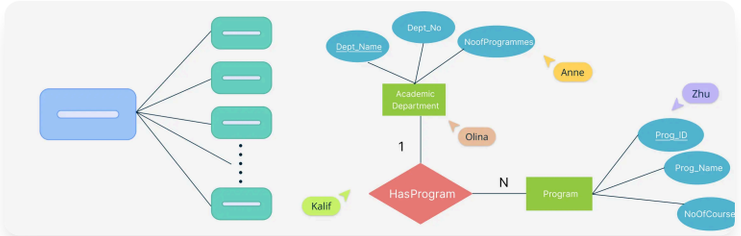
- HasOne/WithMany para definir relacionamentos
- HasForeignKey para especificar a coluna FK
- Convenções automáticas criam FK por nome (TipoId)

## Comportamento de deleção

- Cascade: Remove entidades dependentes automaticamente
- Restrict: Impede exclusão se houver dependentes
- SetNull: Define FK como nula (requer coluna nullable)

## Carregamento de dados

- Eager: Include/ThenInclude (evita problema N+1)
- Explicit: Load/LoadAsync no Entry após consulta
- Lazy: Proxies que carregam sob demanda (com cuidado)




### Dicas para Performance:

- Prefira projeções (Select) em DTOs
- Use filtros adequados antes de Include
- Crie índices para FKs no banco
- Evite loops que causam N+1 queries
- Para testes, use provider InMemory

# Exercício 4 — Cliente e relacionamento 1:N com Conta

**Objetivo:**

Modelar Cliente e vincular múltiplas Contas por Cliente.

**Passos:**

1. **Criar a entidade Cliente:**
   - Adicionar propriedades básicas (Id, Nome, Documento, etc.)
   - Incluir propriedades de navegação para Contas (ICollection<Conta>)
2. **Definir relacionamento 1:N:**
   - Adicionar ClienteId (FK) na classe Conta
   - Configurar propriedade de navegação Cliente na classe Conta
   - Mapear relacionamento via Data Annotations ou Fluent API
3. **Configurar comportamento de deleção:**
   - Definir DeleteBehavior (Restrict, Cascade, SetNull)
   - Gerar migration com nome descritivo (Add-Migration ClienteContaRelationship)
4. **Atualizar banco e inserir dados:**
   - Aplicar migration (Update-Database)
   - Criar endpoints para inserir Clientes
   - Modificar endpoint de criação de Contas para aceitar ClienteId
5. **Implementar consulta por Cliente:**
   - Criar endpoint GET /api/clientes/{id}/contas
   - Utilizar Include para carregar Contas do Cliente (Eager Loading)
   - Adicionar ordenação e filtragem opcional

**Critérios de Aceite:**

- Tabelas Cliente e Conta criadas com relacionamento correto
- Endpoint de listagem de Contas por Cliente funcional
- Operações CRUD mantêm integridade referencial

# Padrão Repository (Introdução)

## Motivação

- Abstrair acesso a dados do restante da aplicação
- Facilitar a testabilidade do código com mocks
- Encapsular queries e regras de persistência complexas

## Quando usar

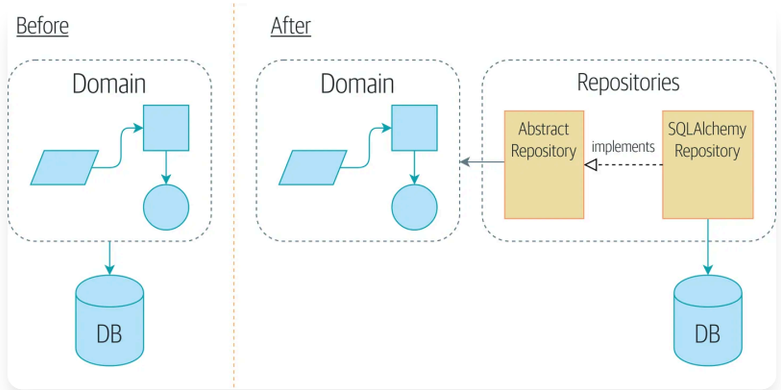
- Projetos com separação clara de camadas
- Múltiplas fontes de dados (não só EF Core)
- Quando a padronização do acesso a dados é prioridade

## Benefícios

- Isolamento da camada de acesso a dados
- Código mais testável (injeção de repositórios mockados)
- Centralização de queries e lógicas específicas por entidade

## Cuidados

- Evitar abstrair recursos úteis do EF (Include, IQueryable)
- Não cair na tentação de repositórios genéricos excessivos
- Alinhar ciclo de vida com Unit of Work (DbContext)




**Integração com DI:**

- Registre com lifetime Scoped (mesmo do DbContext)
- Injete interfaces, não implementações
- Configure no Program.cs: services.AddScoped()
- Repository depende de DbContext: injetado automaticamente

# Exercício 5 — Padrão Repository para Conta

**Objetivo:**

Abstrair o acesso a dados de Conta com um repositório e injetá-lo no controller.

**Passos:**

1. **Criar a interface IContaRepository:**
   - Definir métodos assíncronos (GetAllAsync, GetByIdAsync, etc.)
   - Adicionar consultas específicas (ex: GetByClienteIdAsync)
   - Incluir métodos para criação, atualização e deleção
2. **Implementar a classe ContaRepository:**
   - Injetar e utilizar BankContext internamente
   - Implementar todos os métodos da interface
   - Aplicar Include/ThenInclude quando necessário
3. **Registrar no contêiner de DI:**
   - Adicionar services.AddScoped<IContaRepository, ContaRepository>() no Program.cs
   - Usar lifetime Scoped para acompanhar o ciclo de vida do DbContext
4. **Modificar o ContasController:**
   - Remover a injeção direta de BankContext
   - Injetar e utilizar apenas IContaRepository
   - Adaptar todos os endpoints para usar os métodos do repositório
5. **Validar e testar:**
   - Criar teste unitário com mock do repositório
   - Implementar teste de integração com EF Core InMemory ou SQLite
   - Validar funcionamento dos endpoints via Postman/Insomnia

**Critérios de Aceite:**

- Controller desacoplado do DbContext (depende apenas de IContaRepository)
- Todos os endpoints funcionando corretamente com a nova abstração
- Ao menos um método coberto com teste utilizando provider InMemory