FMX Academy

INFORMAÇÕES			
MÓDULO		**AULA**	**INSTRUTOR**
05	**ASP.NET Core Web API – Fundamentos**	**08 - Revisão**	**Guilherme Paracatu**

1. ASP.NET Core Web API – Fundamentos - Revisão

Objetivo: Revisar os conceitos essenciais sobre APIs REST, Entity Framework Core, Arquitetura e boas práticas abordados nesse módulo, preparando para a avaliação.

1. O Que São APIs e Por Que Usamos REST?

- **API (Interface de Programação de Aplicações):** Pensem no **Garçom Digital**. É o contrato que define como diferentes softwares (frontend, backend, mobile) conversam entre si. Ele abstrai a complexidade interna do servidor (a "cozinha").
- **REST (Representational State Transfer):** Não é um protocolo, mas um **Estilo Arquitetural**. São as "regras de etiqueta" para construir APIs na web que sejam **escaláveis**, **flexíveis** e **padronizadas**, assim como a própria internet. Uma API que segue essas regras é **RESTful**.
- **Conceitos Chave REST:**
  - **Recursos:** Tudo é um recurso (Cliente, Produto, Tarefa) identificado por uma **URI** (geralmente uma URL). Use **substantivos** na URI (ex: /api/tarefas).
  - **Representações:** Interagimos com "fotos" (representações) do estado do recurso, geralmente em **JSON**.
  - **HTTP:** O REST usa os métodos (verbos) e códigos de status do HTTP como base da comunicação.

2. Os Pilares REST Mais Importantes

- **Separação Cliente-Servidor:** Essencial! Permite que frontend e backend evoluam de forma independente.
- **Stateless (Sem Estado): CRUCIAL PARA ESCALABILIDADE!** O servidor não guarda informações de sessão do cliente. Cada requisição contém tudo o que o servidor precisa saber. Permite balanceamento de carga fácil.
- **Interface Uniforme:** A "língua universal" da API. Usamos:
  - **URIs** para identificar os recursos (substantivos).
  - **Verbos HTTP** para definir a ação (CRUD):
    - GET: Ler dados.
    - POST: Criar um novo recurso.
    - PUT: Substituir um recurso **inteiro**. (Cuidado com a perda de dados!).
    - DELETE: Remover um recurso.
    - PATCH (Menos comum): Atualizar *parcialmente* um recurso.

- **Códigos de Status HTTP:** O feedback do servidor:
  - 2xx (Sucesso): 200 OK, 201 Created, 204 No Content.
  - 4xx (Erro do Cliente): 400 Bad Request, 401 Unauthorized, 404 Not Found.
  - 5xx (Erro do Servidor): 500 Internal Server Error.

3. Entity Framework Core (EF Core) 8: A Ponte com o Banco

- **ORM (Object-Relational Mapper):** É o nosso "tradutor" C# <-> SQL. Mapeia classes para tabelas.

- **DbContext (Ex: TarefaContext):** A "central de comando". Gerencia a conexão, mapeia DbSet<T> para tabelas, traduz LINQ para SQL e rastreia alterações (SaveChanges()).

- **Database-First:** O banco de dados já existe. Nós criamos/adaptamos nossas classes C# para espelhá-lo.

- **Mapeamento (Como conectar C# ao Banco):**
  - **Data Annotations ([Table], [Column], [Key]):** Simples, mas "sujam" a entidade.
  - **Fluent API (OnModelCreating no DbContext): Preferível!** Centraliza toda a configuração, mais poderosa (relacionamentos complexos, índices), mantém as entidades limpas.

- **Relacionamentos:**
  - **1:1 (Ex: Tarefa <-> DetalhesTarefa):** Configurado na Fluent API (HasOne, WithOne, HasForeignKey).
  - **1:N (Ex: Usuario -> Tarefa):** Configurado na Fluent API (HasMany, WithOne, HasForeignKey).
  - **N:N com Payload (Ex: Tarefa <-> Tag via TarefaTag):** Requer uma entidade de junção (TarefaTag) e é configurado como dois relacionamentos 1:N na Fluent API.

- **Performance (N+1 vs Eager Loading):**
  - **Problema N+1:** 1 consulta para a lista + N consultas para dados relacionados (causado por Lazy Loading em loops). **Vilão da performance!**

  - **Solução (Eager Loading):** Use .Include(x => x.PropriedadeDeNavegacao) para trazer dados relacionados na mesma consulta (gera JOIN no SQL). **Essencial!**

4. DTOs e AutoMapper: Contratos de API e Mapeamento Eficiente

- **Por Que DTOs (Data Transfer Objects)?**
  - **NUNCA exponha Entidades EF Core diretamente na API!**
  - **Resolve Referência Cíclica:** Evita loops na serialização JSON.
  - **Segurança:** Esconde dados internos do banco.
  - **Contrato Estável:** Desacopla a API da estrutura do banco.

- **AutoMapper:** Biblioteca para **automatizar** o mapeamento Entidade <-> DTO.
  - **Configuração:** Crie um Profile (CreateMap<Origem, Destino>();).
  - **Uso:** Injete IMapper e use _mapper.Map<Destino>(origem).
  - **Poder:** Mapeia propriedades com nomes diferentes (ForMember), valores calculados, etc.

5. Arquitetura e Validação

- **Arquitetura em Camadas:**
  - **Controller:** Recebe HTTP, chama o Service, retorna HTTP. Deve ser "magro".

- **Service:** Contém a **lógica de negócio**, orquestra chamadas ao Repository, mapeia DTOs.
- **Repository:** Responsável **apenas** pelo acesso aos dados (interage com o DbContext).

- **Injeção de Dependência (DI):** Essencial para **desacoplar** as camadas. Usamos **Interfaces** (IService, IRepository) e o container do ASP.NET Core (builder.Services.AddScoped<>) para injetar as dependências via construtor. Baseado no princípio **DIP** do SOLID.

- **Validação:**
  - **Validação de Formato/Estrutura:** Use **Data Annotations** ([Required], [StringLength], etc.) nos **DTOs**. O [ApiController] valida isso **automaticamente** e retorna 400 Bad Request.
  - **Validação de Negócio:** Faça na **camada de Serviço** (ex: verificar se um e-mail já existe no banco).

6. Dicas para a Avaliação Prática (Tarefa API)

- **Estrutura:** Siga a estrutura de pastas solicitada.
- **Fluent API:** Use o OnModelCreating para todo o mapeamento.
- **DI e Interfaces:** Injete interfaces, não classes concretas.
- **DTOs e AutoMapper:** Use DTOs para todas as entradas e saídas do Controller. Configure o AutoMapper(Opcional).
- **Entrega Mínima (Dia da Prova):** Foco em ter a estrutura pronta e os GET/POST de Tarefas funcionando.

Swagger: Teste cada endpoint à medida que o desenvolve!