Desenvolvimento para *Internet* II

Outras ações na API e detalhes do Entity Framework

Prof. Dr. Tiago Alexandre Dócusse

tad@ifsp.edu.br

Introdução

Na aula anterior vimos como criar a API, funções básicas para Salvar e Listar dados, bem como conectar na base de dados com o Entity Framework.

Nesta aula vamos ver o restante das ações CRUD, bem como mais detalhes do Entity Framework.

Origem dos dados

O código do método POST da aula anterior é este:

```
[HttpPost]
public ActionResult Post(TipoCurso item)
{
    try
    {
        context.TipoCursos.Add(item);
        context.SaveChanges();
        return Ok("Tipo de curso salvo com sucesso");
    }
    catch
    {
        return BadRequest("Erro ao salvar o tipo de curso informado");
    }
}
```

No código anterior, perceba que não informamos de onde, na solicitação, a *model* pode ser informada (no caso do nosso exemplo, do corpo da solicitação).

Podemos utilizar atributos para definir a origem dos parâmetros informados para uma ação, de forma a aumentar a segurança do nosso código e evitar que os dados sejam lidas de uma origem que não esperamos.

As origens mais utilizadas de dados são:

Nome do parâmetro	Origem do parâmetro
FromBody	Corpo da solicitação
FromForm	Dados do formulário no corpo da solicitação
FromHeader	Cabeçalho da solicitação
FromQuery	Da URL da solicitação
FromRoute	Da rota utilizada na solicitação

Devemos utilizar estes atributos antes do parâmetro, entre colchetes.

Como o nosso parâmetro deve ser enviado pelo corpo da solicitação, o método ficará da seguinte forma:

```
[HttpPost]
public ActionResult Post([FromBody]TipoCurso item)
{
    try
    {
        context.TipoCursos.Add(item);
        context.SaveChanges();
        return Ok("Tipo de curso salvo com sucesso");
    }
    catch
    {
        return BadRequest("Erro ao salvar o tipo de curso informado");
    }
}
```

Ações assíncronas

As ações realizadas até agora são todas síncronas, ou seja, uma ação só é executada quando a última ação antes dela terminar de executar.

Ações assíncronas removem esta necessidade, o que nos auxilia em algumas situações. Por exemplo, ao abrir um site de notícias, devemos carregar as notícias, as mensagens que o usuário recebeu, as informações do usuário, etc.

Ao invés de realizar três requisições síncronas, fazendo com que uma espera a outra terminar para poder executar, podemos fazer essas três requisições de forma assíncrona, e preencher o *site* à medida que elas terminem de executar, independentes umas das outras.

Para podermos utilizar ações assíncronas na API, é necessário que elas sejam definidas como assíncronas.

O retorno de uma função assíncrona deve ser do tipo `Task<>`. Normalmente usamos `Task<ActionResult<...>>`.

Mesmo que a ação não realize nenhuma ação assíncrona, é interessante já modelar as ações como assíncronas para uma possível utilização de método assíncrono.

Em métodos assíncronos que devemos aguardar a execução de alguma instrução, devemos utilizar o operador `await` antes da chamada de um método assíncrono.

Ação POST assíncrona:

```
[HttpPost]
public async Task<ActionResult> Post([FromBody]TipoCurso item)
{
    try
    {
        await context.TipoCursos.AddAsync(item);
        await context.SaveChangesAsync();
        return Ok("Tipo de curso salvo com sucesso");
    }
    catch
    {
        return BadRequest("Erro ao salvar o tipo de curso informado");
    }
}
```

Ação GET assíncrona:

```
[HttpGet]
public async Task<ActionResult<IEnumerable<TipoCurso>>> Get()
{
    try
    {
        return Ok(await context.TipoCursos.ToListAsync());
    }
    catch
    {
        return BadRequest("Erro ao listar os tipos de curso");
    }
}
```

Pode ser necessário incluir a seguinte linha no início do arquivo:

```
using Microsoft.EntityFrameworkCore;
```

Ação para obtenção de item pelo Id

Vamos criar uma ação GET para retornar um tipo de curso especificado pelo seu Id.

Como já temos uma ação GET na rota padrão (perceba que não definimos um valor de rota para o método que lista todos os tipos de curso), vamos dizer que esta nova ação será definida pela rota `/id` em uma solicitação do tipo GET.

Usaremos alguns métodos novos:

- `Any` ou `AnyAsync`: retorna verdadeiro se um elemento satisfaz uma condição, falso caso contrário.
- `Find` ou `FindAsync`: retorna um objeto através da sua chave primária.

O código da ação de obtenção pelo Id ficará assim:

```
[HttpGet("{id}")]
public async Task<ActionResult<TipoCurso>> Get([FromRoute] int id)
{
    try
    {
        if (await context.TipoCursos.AnyAsync(p => p.Id == id))
            return Ok(await context.TipoCursos.FindAsync(id));
        else
            return NotFound("O tipo de curso informado não foi encontrado");
    }
    catch
    {
        return BadRequest("Erro ao efetuar a busca de tipo de curso");
    }
}
```

Verifique o funcionamento acessando a rota `/api/TipoCurso/1`, por exemplo.

Ação de atualização

Para a ação de atualização, utilizaremos o método PUT ao invés de PATCH.

Por definição, o método PUT espera que todos os dados sejam informados.

Não é obrigatório, mas é comum informarmos uma rota para atualização de um item específico. Assim, podemos utilizar a rota sem argumento do PUT para alguma outra ação que desejarmos.

Também é comum informar dois argumentos: a chave primária, cuja origem deve ser a rota, e a *model* contendo todos os dados no item a ser atualizado, originada do corpo da requisição.

Cuidado: é obrigatório informar o id no objeto enviado. Caso ele esteja nulo, pode acontecer (dependendo da programação) de ser inserido um novo item no repositório.

```
[HttpPut("{id}")]
public async Task<ActionResult> Put([FromRoute] int id, [FromBody] TipoCurso model)
{
    if (id != model.Id) //se é diferente da rota, erro
        return BadRequest("Tipo de curso inválido");

    try
    {
        //se não existe, erro, senão cria um novo tipo de curso
        if (!await context.TipoCursos.AnyAsync(p => p.Id == id))
            return NotFound("Tipo de curso inválido");

        context.TipoCursos.Update(model);
        await context.SaveChangesAsync();
        return Ok("Tipo de curso salvo com sucesso");
    }
    catch
    {
        return BadRequest("Erro ao salvar o tipo de curso informado");
    }
}
```

É comum cometermos o seguinte erro ao tentar atualizar um item:

```
//Exemplo de código errado apenas para informação. Não usar esta implementação.
[HttpPut("{id}")]
public async Task<ActionResult> Put([FromRoute] int id, [FromBody] TipoCurso model)
{
    if (id != model.Id)
        return BadRequest();

    try
    {
        TipoCurso antigo = await context.TipoCursos.FindAsync(id);
        if (antigo != null)
        {
            context.TipoCursos.Update(model);
            await context.SaveChangesAsync();
            return Ok("Tipo de curso salvo com sucesso, mas da forma errada");
        }
        else
            return NotFound();
    }
    catch (Exception e)
    {
        return BadRequest(e.Message);
    }
}
```

Quando executamos o método `FindAsync`, ele vincula o objeto `antigo` ao id pesquisado, e apenas ele poderá ser utilizado nesta transação (a não ser que ele seja liberado).

Quanto tentamos atualizar o objeto `model`, ocorre um erro, pois o id de `model` está vinculado ao objeto `antigo`.

Neste caso, é recomendado programar como no exemplo anterior, mas se precisarmos liberar objetos vinculados, o fazemos com o comando:

```
context.ChangeTracker.Clear();
```

Ação de Remoção

Na ação de remoção, informamos para a requisição apenas a chave primária do objeto a ser removido.

Utilizamos o método Remove para informar o objeto a ser removido. Portanto, devemos buscar o objeto no repositório.

Vamos utilizar como parte da rota a chave primária, para deixar a rota geral disponível para alguma outra ação que deseje a utilizar.

```
[HttpDelete("{id}")]
public async Task<ActionResult> Delete([FromRoute] int id)
{
    try
    {
        TipoCurso model = await context.TipoCursos.FindAsync(id);

        if (model == null)
            return NotFound("Tipo de curso inválido");

        context.TipoCursos.Remove(model);
        await context.SaveChangesAsync();
        return Ok("Tipo de curso removido com sucesso");
    }
    catch
    {
        return BadRequest("Falha ao remover o tipo de curso");
    }
}
```

Pesquisa por igualdade de um atributo específico

Vamos criar uma ação para retornar todos os tipos de curso com um nome específico.

Como já usamos a rota GET padrão para listar todos os tipos de curso e a rota com um argumento para buscar por um identificador específico, esta ação ficará na rota `/api/TipoCurso/pesquisaNome`.

O nome do curso a ser pesquisado será informado através da rota. Por exemplo: `/api/TipoCurso/pesquisaNome/superior`.

Usaremos o método `Where`, que retorna uma lista de entidades que satisfazem sua condição.

```
[HttpGet("pesquisaNome/{nome}")]
public async Task<ActionResult<IEnumerable<TipoCurso>>> Get([FromRoute] string nome)
{
    try
    {
        List<TipoCurso> resultado = await context.TipoCursos.Where(p => p.Nome == nome).ToListAsync();
        return Ok(resultado);
    }
    catch
    {
        return BadRequest("Falha ao buscar um tipo de curso");
    }
}
```

Acesse, por exemplo, a rota `/api/TipoCurso/pesquisaNome/superior` para pesquisar tipos de curso que possuem nome idêntico à palavra `superior`.

Pesquisa por semelhança de um atributo específico

Nesta ação, o nome é pesquisado por semelhança ao invés de igualdade.

Ela estará disponível através da rota `/api/TipoCurso/pesquisaNomeSemelhante`.

Como já temos um método que recebe apenas uma *string* como entrada, devemos criar um método com nome diferente.

Utilizaremos o método `Contains` da classe `String` para fazer esta comparação.

```
[HttpGet("pesquisaNomeSemelhante/{nome}")]
public async Task<ActionResult<IEnumerable<TipoCurso>>> PesquisaNomeSemelhante([FromRoute] string nome)
{
    try
    {
        List<TipoCurso> resultado = await context.TipoCursos.
        Where(p => p.Nome.Contains(nome)).ToListAsync();
        return Ok(resultado);
    }
    catch
    {
        return BadRequest("Falha ao buscar um tipo de curso");
    }
}
```

Pesquisa por vários atributos

Vamos fazer a pesquisa por vários atributos de um objeto informado no corpo da requisição.

Devemos usar um método POST devido ao objeto ser informado no corpo da requisição.

Esta ação estará na rota `/api/TipoCurso/Pesquisa`.

Como não vamos passar informação pela rota, informamos a rota através do atributo `[Route]`.

Devemos pesquisar por todo atributo da *model* que for diferente de nulo.

Poderíamos tentar fazer uma ação de acordo com o código a seguir, mas esta abordagem possui um erro.

```
//Exemplo de código errado apenas para informação. Não utilize esta implementação.
[Route("pesquisa")]
[HttpPost]
public async Task<ActionResult<IEnumerable<TipoCurso>>> Pequisa([FromBody] TipoCurso model)
{
    List<TipoCurso> resultado = null;

    // todo: realizar a verificação apenas de acordo com os atributos que são diferentes de nulo

    return Ok(resultado);
}
```

Se não pesquisarmos por ambos nome e descrição, veremos a mensagem que eles são obrigatórios, já que os definimos assim na *model*.

Para resolver este problema, podemos escrever muitas linhas de código para poder habilitar uma ação para ignorar as restrições de modelagem.

Ou podemos mudar o argumento do método para um objeto do tipo `object`, o que burla a validação da Controller.

Tome cuidado ao usar esta abordagem, pois não há garantias de que as restrições serão validadas, sendo necessário mais código para fazer essas validações.

Transforme o objeto recebido para um objeto do tipo `TipoCurso`:

```
//Exemplo de código errado apenas para informação. Não utilize esta implementação.
[Route("pesquisa")]
[HttpPost]
public async Task<ActionResult<IEnumerable<TipoCurso>>> Pesquisa([FromBody] object item)
{
    TipoCurso model = JsonSerializer.Deserialize<TipoCurso>(item.ToString());

    // Restante do código será feito depois.
    // Este é apenas um exemplo para mostrar que a validação da controller não será executada.
    await context.SaveChangesAsync();
    return new List<TipoCurso>();
}
```

Feito isso, as restrições da *model* não serão validadas.

Agora temos um segundo problema: para nome e descrição, temos quatro possibilidades:

- Nome ≠ nulo, Descrição ≠ nulo
- Nome ≠ nulo, Descrição = nulo
- Nome = nulo, Descrição ≠ nulo
- Nome = nulo, Descrição = nulo

Nosso código ficaria assim:

```
//Código ineficiente apenas para demonstração. Não utilize esta implementação.
[Route("pesquisa")]
[HttpPost]
public async Task<ActionResult<IEnumerable<TipoCurso>>> Pesquisa([FromBody] object item)
{
    try
    {
        TipoCurso model = JsonSerializer.Deserialize<TipoCurso>(item.ToString());

        List<TipoCurso> resultado = null;

        if (model.Nome != null && model.Descricao != null)
            resultado = await context.TipoCursos.Where(p => p.Nome == model.Nome && p.Descricao == model.Descricao).ToListAsync();
        else if (model.Nome != null)
            resultado = await context.TipoCursos.Where(p => p.Nome == model.Nome).ToListAsync();
        else if (model.Descricao != null)
            resultado = await context.TipoCursos.Where(p => p.Descricao == model.Descricao).ToListAsync();
        else
            resultado = await context.TipoCursos.ToListAsync();

        return Ok(resultado);
    }
    catch
    {
        return BadRequest();
    }
}
```

Caso tenhamos 3 atributos para pesquisar, teríamos 8 possibilidades:

- Nulo, Nulo, Nulo
- Nulo, Nulo, Não Nulo
- Nulo, Não Nulo, Nulo
- Nulo, Não Nulo, Não Nulo
- Não Nulo, Nulo, Nulo
- Não Nulo, Nulo, Não Nulo
- Não Nulo, Não Nulo, Nulo
- Não Nulo, Não Nulo, Não Nulo

Ou seja, 2^n comandos, sendo n a quantidade de atributos. Para cinco atributos, teríamos 32 instruções `if-else`, o que não é muito viável de realizar.

Outra opção é criar um método para fazer tal procedimento. Primeiro, criamos uma classe auxiliar em `/Data`:

```
using System.Linq.Expressions;

public static class LinqExtensions
{
    public static IQueryable<TSource> WhereIf<TSource>(this IQueryable<TSource> source, bool condition, Expression<Func<TSource, bool>> predicate)
    {
        return condition ? source.Where(predicate) : source;
    }
}
```

E utilizamos o método `WhereIf` criado toda vez que quisermos adicionar um if e uma função lambda.

Agora se tivermos *n* atributos, a quantidade de comandos será *n*.

Nosso método de pesquisa ficará assim:

```
[Route("pesquisa")]
[HttpPost]
public async Task<ActionResult<IEnumerable<TipoCurso>>> Pesquisa([FromBody] object item)
{
    try
    {
        TipoCurso model = JsonSerializer.Deserialize<TipoCurso>(item.ToString());

        List<TipoCurso> resultado = await context.TipoCursos
            .WhereIf(model.Nome != null, p => p.Nome == model.Nome)
            .WhereIf(model.Descricao != null, p => p.Descricao == model.Descricao).ToListAsync();

        return Ok(resultado);
    }
    catch
    {
        return BadRequest();
    }
}
```

Outros conceitos e recursos do Entity Framework

Até agora seguimos algumas convenções do Entity Framework.

Para fazer a modelagem diferente dessas convenções, podemos usar os atributos no *namespace* `DataAnnotations`, ou sobrescrever algumas funções da classe `DataContext`, criada em `Data`, ou usar uma linguagem chamada Fluent API.

Se houver mais de uma mesma configuração para um item, a ordem de prioridade (da maior para a menor) é:

- Fluent API
- Data Annotation
- Convenção do EF

https://learn.microsoft.com/en-us/ef/core/modeling/

Tabelas

Toda propriedade pública em `DataContext` do tipo `DbSet<>` é mapeada para uma tabela na base de dados.

Caso queira que uma propriedade deste tipo não seja mapeada para uma tabela na base de dados, podemos usar o atributo `NotMapped` antes da propriedade:

```
using System.ComponentModel.DataAnnotations.Schema;

[NotMapped]
public class Curso
{
    public int? Id {get;set;}
}
```

Mesmo que haja a propriedade `DbSet<Curso>` em `DataContext`, ela não será mapeada para uma tabela na base de dados.

Usando a `Fluent API`, fazemos uma sobrecarga do método `OnModelCreating` na classe `DataContext`, e usamos o método `Ignore` do objeto utilizado como argumento:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<Curso>();
}
```

Nome das tabelas

A convenção do EF utiliza o nome da propriedade `DbSet<>` em `DataContext` como o nome da tabela mapeada na base de dados.

Para mudar esse valor com `DataAnnotation`, utilize o atributo `Table` antes do nome da classe:

```
using System.ComponentModel.DataAnnotations.Schema;

[Table("TBCurso")]
public class Curso
{
    public int? Id {get;set;}
}
```

Para mudar esse valor com `Fluent API`, na sobrecarga do método `OnModelCreating` de `DataContext`, utilize o método `ToTable` para a entidade desejada:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().ToTable("TBCursos");
}
```

Para adicionar um comentário à tabela, utilize `Comment` antes da sua definição.

```
using Microsoft.EntityFrameworkCore;

[Comment("Curso de uma instituição de ensino")]
public class Curso
{
    public int? Id {get;set;}
}
```

Ou usando Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().ToTable(p => p.HasComment("Curso de uma instituição de ensino"));
}
```

Mapeamento das colunas

Toda propriedade pública será mapeada para uma coluna de uma tabela.

Caso queira que uma propriedade não seja mapeada, utilize o atributo `NotMapped` antes de seu nome:

```
using System.ComponentModel.DataAnnotations.Schema;

public class Curso
{
    public int? Id {get;set;}

    [NotMapped]
    public string NomeTipoCurso {get;set;}
}
```

Usando a Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Ignore(p => p.NomeTipoCurso);
}
```

Nome das colunas

Por padrão, o nome da propriedade será utilizado como o nome da coluna.

Caso queira alterar, utilize o atributo `Column`:

```
using System.ComponentModel.DataAnnotations.Schema;

public class Curso
{
    public int? Id {get;set;}

    [Column("NomeCurso")]
    public string Nome {get;set;}
}
```

Ou com Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.Nome).HasColumnName("NomeCurso");
}
```

Tipos das colunas

O tipo de dados utilizado na coluna é definido de acordo com o tipo de dados utilizado na *model*.

Caso queira alterar, utilize a propriedade `TypeName` do atributo `Column`:

```
using System.ComponentModel.DataAnnotations.Schema;

public class Curso
{
    public int? Id {get;set;}

    [Column(TypeName = "varchar(300)")]
    public string Nome {get;set;}
}
```

Ou com a Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>
    (
        e => {
            e.Property(p => p.Nome).HasColumnType("varchar(300)");
        }
    );
}
```

Obrigatoriedade da coluna

Caso uma propriedade seja de um tipo não anulável, ela será obrigatória na modelagem. Propriedades de tipos anuláveis são opcionais.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Curso
{
    public int? Id {get;set;}

    public string Nome {get;set;} //campo obrigatório

    public string? Descrição {get;set;} //campo opcional
}
```

Comentários de colunas

Comentários de colunas podem ser feitos com o atributo `Comment`:

```
using Microsoft.EntityFrameworkCore;

public class Curso
{
    public int? Id {get;set;}

    [Comment("Nome do curso")]
    public string Nome {get;set;}

    public string? Descrição {get;set;}
}
```

Ou com Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.Descrição).HasComment("Uma breve descrição do curso.");
}
```

Chave primária

Por padrão, caso a *model* possua uma propriedade chamada `Id` ou `NomeModelId`, esta será a sua chave primária.

Caso a chave primária tenha um nome diferente, utilize o atributo `Key` para definir como uma chave primária:

```
using System.ComponentModel.DataAnnotations;

public class Curso
{
    [Key]
    public string Nome {get;set;}
}
```

Ou com Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().HasKey(p => p.Nome);
}
```

Caso você queira usar uma chave composta, onde duas ou mais propriedades fazem parte da chave, utilize o atributo `PrimaryKey`:

```
using Microsoft.EntityFrameworkCore;

[PrimaryKey(nameof(Nome), nameof(AnoOferta))]
public class Curso
{
    public string Nome {get;set;}
    public int AnoOferta {get;set;}
}
```

Ou com Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().HasKey(p => new {p.Nome, p.AnoOferta});
}
```

Colunas com valor padrão

Para definir um valor padrão para uma propriedade, use Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.Ano).HasDefaultValue(2023);
}
```

Também é possível definir o valor padrão como o retorno de um código SQL:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.Ano).HasDefaultValueSql("select year(getdate())");
}
```

Colunas auto-incrementáveis

Por padrão, toda chave primária inteira que não é composta é definida como um campo auto incrementável (Identidade, no SQL Server).

Caso queira remover este comportamento, utilize o atributo `DataBaseGenerated`:

```
using System.ComponentModel.DataAnnotations.Schema;

public class Curso
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int? Id {get;set;}
}
```

Ou com Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.Id).ValueGeneratedNever();
}
```

Para propriedades que não são chaves primárias, também é possível marcá-las como auto incremento.

No entanto, cada *model* pode possuir apenas 1 propriedade identidade, incluindo a chave primária.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

public class Curso
{
    [Key]
    public string Chave {get;set;}

    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int? CampoIncrementavel {get;set;}
}
```

Com Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.CampoIncrementavel).ValueGeneratedOnAdd();
}
```

O código anterior inicia o campo identidade com valor inicial 1 e incremento em 1. Para usar valor inicial 2 e incremento 5, por exemplo, utilize:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().Property(p => p.CampoIncrementavel).UseIdentityColumn(2, 5);
}
```

Índices

Índices podem ser criados utilizando a propriedade `Index` antes do nome da classe.

Por exemplo, para criar um índice referente à propriedade Nome, podemos fazer desta forma com `DataAnnotation`:

```
using Microsoft.EntityFrameworkCore;

[Index(nameof(Nome))]
public class Curso
{
    public int Id {get;set;}

    public string Nome {get;set;}
}
```

Ou então, com a Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().HasIndex(p => p.Nome);
}
```

Para índices compostos, basta acrescentar mais argumentos à propriedade Index:

```
using Microsoft.EntityFrameworkCore;

[Index(nameof(Nome), nameof(Tipo))]
public class Curso
{
    public int Id {get;set;}

    public string Nome {get;set;}

    public string Tipo {get;set;}
}
```

Com a Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().HasIndex(p => new {p.Nome, p.Tipo});
}
```

Para criar índices únicos, utilize o argumento `IsUnique` do atributo Index:

```
using Microsoft.EntityFrameworkCore;

[Index(nameof(Nome), IsUnique = true)]
public class Curso
{
    public int Id {get;set;}

    public string Nome {get;set;}
}
```

Ou com a Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Curso>().HasIndex(p => p.Nome).IsUnique();
}
```

Consultas com o Entity Framework

Para realizar consultas com o Entity Framework, vamos utilizar um Context (como visto na aula anterior).

Por exemplo, para armazenar um objeto `tipo` em TipoCurso, fazemos:

```
context.TipoCursos.Add(tipo);
await context.SaveChangesAsync();
```

Já para trazer todas as instâncias de tipos de curso:

```
context.TipoCursos.ToListAsync();
```

Para filtrar dados, utilizamos o operador `Where`:

```
await context.TipoCursos.Where(p => p.Nome == "Pos").ToListAsync();
```

Para ordenar dados em ordem crescente, utilizamos o operador `OrderBy`:

```
await context.TipoCursos.Where(p => p.Nome == "Pos").OrderBy(p => p.Nome).ToListAsync();
```

Para ordenar em ordem decrescente, utilizamos o operador `OrderByDescending`:

```
await context.TipoCursos.Where(p => p.Nome == "Pos").OrderByDescending(p => p.Nome).ToListAsync();
```

Parar ordenar por um próximo parâmetro, em ordem crescente, utilizamos o operador `ThenBy`:

```
await context.TipoCursos.OrderBy(p => p.Nome).ThenBy(p => p.Id).ToListAsync();
```

Para ordenar por um próximo parâmetro, em ordem decrescente, utilizamos o operador `ThenByDescending`:

```
await context.TipoCursos.OrderBy(p => p.Nome).ThenByDescending(p => p.Id).ToListAsync();
```

Paginação

Para buscar apenas os primeiros `n` registros de uma consulta, utilize o operador `Take`:

```
int n = 10;
await context.TipoCursos.Take(n).ToListAsync();
```

Para ignorar os `z` primeiros registros e buscar os próximos `n` registros de uma consulta, utilize o operador `Skip` em conjunto com o operador `Take`:

```
int n = 10;
int z = 30;
await context.TipoCursos.Skip(z).Take(n).ToListAsync();
```

O problema da abordagem anterior é que, caso dados sejam alterados ou inseridos entre duas consultas subsequentes, pode haver repetição de dados já recuperados ou não exibição de dados não recuperados.

Caso esteja utilizando um identificador na *model*, é possível utilizar o operador `Where` para filtrar `n` registros a partir do último registro recuperado:

```
int ultimoId = 20;
int n = 10;
await context.TipoCursos.Where(p => p.Id > ultimoId).Take(n).ToListAsync();
```

Para fazer consultas SQL com o texto explícito, utilize o operador `FromSql`:

```
var nome = "Pos";
await context.TipoCursos.FromSql($"SELECT Id, Nome from TipoCursos where Nome = {nome}").ToListAsync();
```

É possível utilizar os outros operadores com qualquer operador que retorne uma coleção de dados.

Relacionamentos

Para relacionamentos um para muitos, vamos considerar nosso exemplo atual: `Cursos` e `Tipos de Cursos` em uma instituição de ensino.

Neste exemplo:

- um curso pode ser categorizado como de apenas um tipo de curso;
- um tipo de curso pode ter vários cursos vinculados a ele.

Já temos a Model referente a um tipo de curso:

```
using System.ComponentModel.DataAnnotations;

public class TipoCurso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    [MinLength(3)]
    [MaxLength(100, ErrorMessage = "O nome deve possuir, no máximo, 100 caracteres")]
    public string Nome { get; set; }

    [Required]
    [MinLength(5, ErrorMessage = "A descrição deve possuir, no mínimo, 5 caracteres")]
    [MaxLength(100)]
    public string Descricao { get; set; }
}
```

Crie o arquivo `Models/Curso.cs` com uma primeira versão da Model de curso:

```
using System.ComponentModel.DataAnnotations;

public class Curso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    [MinLength(3)]
    [MaxLength(100, ErrorMessage = "O nome deve possuir, no máximo, 100 caracteres")]
    public string Nome { get; set; }
}
```

Para definir o relacionamento, adicionamos na classe `Curso` uma propriedade do tipo `int` com o nome da tabela principal seguida de Id, ou seja, `TipoCursoId`. Essa será a chave estrangeira do relacionamento.

Dessa forma, ao fazer uma recuperação de um item `Curso`, a propriedade TipoCursoId será preenchida com o valor correspondente.

```
using System.ComponentModel.DataAnnotations;

public class Curso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    [MinLength(3)]
    [MaxLength(100, ErrorMessage = "O nome deve possuir, no máximo, 100 caracteres")]
    public string Nome { get; set; }

    [Required]
    public int TipoCursoId { get; set; }
}
```

Para obter os dados referentes ao objeto `TipoCurso` vinculado, é possível utilizar o método `Find` com o seu identificador.

No entanto, é possível deixar a modelagem pronta para isso, e realizar um operador posteriormente para buscar esse conteúdo.

Adicione um objeto da classe `TipoCurso` na classe `Curso`.

Não é necessário marcar este objeto como `NotMapped`.

```
using System.ComponentModel.DataAnnotations;

public class Curso
{
    [Required]
    public int Id { get; set; }

    [Required(ErrorMessage = "O nome é obrigatório")]
    [MinLength(3)]
    [MaxLength(100, ErrorMessage = "O nome deve possuir, no máximo, 100 caracteres")]
    public string Nome { get; set; }

    [Required]
    public int TipoCursoId { get; set; }

    public TipoCurso? TipoDoCurso { get; set; }
}
```

Acrescente um `DbSet` para `Curso` em `DataContext`:

```
using Microsoft.EntityFrameworkCore;

public class DataContext : DbContext
{
    public DataContext(DbContextOptions<DataContext> options)
        : base(options) { }

    public DbSet<TipoCurso> TipoCursos { get; set; } = null!;
    public DbSet<Curso> Cursos { get; set; } = null!;
}
```

Crie uma nova migração:

```
dotnet ef migrations add AdicaoCurso
```

Envie a atualização ao SGBD:

```
dotnet ef database update
```

Crie uma Controller para Curso em `/Controllers/CursoController.cs`, com as ações padrão como visto anteriormente.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

[Route("api/[controller]")]
[ApiController]
public class CursoController : ControllerBase
{
    private readonly DataContext context;

    public CursoController(DataContext _context)
    {
        context = _context;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<Curso>>> Get()
    {
        try
        {
            return Ok(await context.Cursos.ToListAsync());
        }
        catch
        {
            return BadRequest("Erro ao listar os cursos");
        }
    }

    [HttpPost]
    public async Task<ActionResult> Post([FromBody] Curso item)
    {
        try
        {
            await context.Cursos.AddAsync(item);
            await context.SaveChangesAsync();
            return Ok("Curso salvo com sucesso");
        }
        catch
        {
            return BadRequest("Erro ao salvar o curso informado");
        }
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<Curso>> Get([FromRoute] int id)
    {
        try
        {
            if (await context.Cursos.AnyAsync(p => p.Id == id))
                return Ok(await context.Cursos.FindAsync(id));
            else
                return NotFound("O curso informado não foi encontrado");
        }
        catch
        {
            return BadRequest("Erro ao efetuar a busca de curso");
        }
    }

    [HttpPut("{id}")]
    public async Task<ActionResult> Put([FromRoute] int id, [FromBody] Curso model)
    {
        if (id != model.Id)
            return BadRequest("Curso inválido");

        try
        {
            if (!await context.Cursos.AnyAsync(p => p.Id == id))
                return NotFound("Curso inválido");

            context.Cursos.Update(model);
            await context.SaveChangesAsync();
            return Ok("Curso salvo com sucesso");
        }
        catch
        {
            return BadRequest("Erro ao salvar o curso informado");
        }
    }

    [HttpDelete("{id}")]
    public async Task<ActionResult> Delete([FromRoute] int id)
    {
        try
        {
            Curso model = await context.Cursos.FindAsync(id);

            if (model == null)
                return NotFound("Curso inválido");

            context.Cursos.Remove(model);
            await context.SaveChangesAsync();
            return Ok("Curso removido com sucesso");
        }
        catch
        {
            return BadRequest("Falha ao remover o curso");
        }
    }
}
```

Crie alguns cursos novos no Postman/Swagger através da rota POST. Se atente para usar um identificador de tipo de curso válido.

Exemplo de objeto para inserção:

```
{
    "Id" : 2,
    "Nome": "ADS",
    "TipoCursoId": 1
}
```

Ao recuperar as informações dos cursos através da ação GET, perceba que o objeto `TipoDoCurso` possui valor `null`, e apenas o seu identificador foi preenchido.

Para recuperar todos os dados do objeto, antes do método `ToListAsync`, devemos utilizar o operador `Include` para informar os dados do objeto vinculado que deseja recuperar.

Para isso, altere a ação `Get()` de `CursoController` para o seguinte código:

```
[HttpGet]
public async Task<ActionResult<IEnumerable<Curso>>> Get()
{
    try
    {
        return Ok(await context.Cursos.Include(p => p.TipoDoCurso).ToListAsync());
    }
    catch
    {
        return BadRequest("Erro ao listar os cursos");
    }
}
```

Ao listar todos os cursos, as informações do tipo de curso devem vir preenchidas.

Para a ação `Get(id)` da mesma Controller, não é possível utilizar o operador `Include` com o método `FindAsync`.

Utilizaremos, então, o método `FirstOrDefault`, que retorna o primeiro elemento da tabela que satisfaz uma condição, ou nulo, caso nenhum elemento satisfaça a condição.

Como condição, utilizaremos a comparação do identificador dos itens da tabela com o identificador informado na rota.

```
[HttpGet("{id}")]
public async Task<ActionResult<Curso>> Get([FromRoute] int id)
{
    try
    {
        if (await context.Cursos.AnyAsync(p => p.Id == id))
            return Ok(await context.Cursos.Include(p => p.TipoDoCurso).FirstOrDefaultAsync(p => p.Id == id));
        else
            return NotFound("O curso informado não foi encontrado");
    }
    catch
    {
        return BadRequest("Erro ao efetuar a busca de curso");
    }
}
```

Perceba agora que o objeto completo do tipo de curso é retornado junto com as informações do curso.