

Resumo - ASP.NET



*Enzo Furegatti Spinella
Nícolas Maisonnète Duarte
2020*

Uma Breve Introdução

Neste material, o objetivo é resumir o conteúdo do ASP.NET passado no 2º ano de Informática do COTUCA. Vale mencionar que é recomendada a leitura da Apostila de ASP.NET da professora Patrícia, para que se tenha uma melhor compreensão.

Startup.cs

Startup.cs é uma classe escrita na linguagem de programação C# que, basicamente, serve para definir as configurações mais básicas do seu programa. Ela é executada a partir da classe Program.cs, que não será abordada neste material por não apresentar grande complexidade e por não requerer mudanças por parte do usuário em vários casos.

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<EscolaContext>
        (
            x =>
x.UseSqlServer(Configuration.GetConnectionString("StringConexaoSQLServer"))
        );
        services.AddControllers();
        services.AddScoped<IRepository, Repository>();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        //app.UseHttpsRedirection();

        app.UseRouting();
    }
}
```

```

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}

```

Podemos observar o método void **ConfigureServices**, onde estabeleceremos qual será o contexto utilizado no nosso programa, qual será a string de conexão - “StringConexaoSQLServer” -, e também definiremos que os controllers serão adicionados pelo método de services chamado **AddControllers**, além do repositório e de sua interface pelo método **AddScoped**. Reiteramos que, no método **AddScoped**, é necessário que os dados estejam na ordem IRepository, e então Repository.

Além disso, é importante mencionar que foi comentado o método do app **UseHttpsRedirection** para que a nossa API possa rodar sem complicações de redirecionamento no navegador, e que utilizaremos o routing (rotas), apresentado pelo método **UseRouting**.

Finalmente, atente-se também à utilização do **AddDbContext**, método de services, pois observaremos na classe EscolaContext.cs a sua presença em forma de herança.

Aluno.cs

Aluno.cs é uma classe escrita na linguagem de programação C# que possui papel parecido com de uma DBO (Data Base Object). Ela, basicamente, define os atributos de “algo”, no caso, um aluno. No exemplo da professora Patrícia, esse aluno é uma tabela do banco de dados, com exatamente os mesmo atributos da classe.

```

public class Aluno
{
    public int Id { get; set; }
    public string RA { get; set; }
    public string Nome { get; set; }
    public int codCurso { get; set; }
}

```

Podemos observar que não há uma declaração explícita sequer de qualquer variável, mas sim uma série do que chamamos de encapsulamentos,

que são getters e setters, juntos em uma mesma **propriedade**. Observação: observe a ausência de parênteses nas **propriedades**, algo que é uma marca registrada delas, o que as diferem de métodos e de variáveis.

EscolaContext.cs

EscolaContext.cs é uma classe escrita na linguagem de programação C# que representa um “contexto” a ser utilizado na nossa API. Mas, o que é esse “contexto”? Basicamente, é uma generalização do que poderemos encontrar na nossa API, no caso, alunos, portanto o contexto será de uma escola! Esse é o tão falado contexto nas análises textuais, como o contexto de um assassinato, onde é pressuposto que se tenha um assassino, uma vítima, e um policial (esperamos).

```
public class EscolaContext : DbContext
{
    public EscolaContext(DbContextOptions<EscolaContext> options) :
base(options)
    {

    }

    public DbSet<Aluno> Alunos { get; set; }
}
```

Podemos observar, como dito no tópico da classe **Startup.cs**, a classe **DbContext**, de onde serão herdados os métodos para o nosso próprio contexto.

Também é possível identificar a criação de uma nova propriedade que retorna ou “seta” um **DbSet** de alunos, que utilizaremos para coletar esses alunos, mas veremos isso mais profundamente em seguida na seção dos repositórios.

IRepository.cs e Repository.cs

IRepository.cs é uma classe escrita na linguagem de programação C# que representa uma interface de um repositório. Temos que criar essa classe por um motivo que pode deixar meio mundo perturbado: ela é necessária para fazer a API funcionar por questão de escolha do criador do .NET CORE. À primeira vista, pode parecer algo desnecessário, mas acaba sendo um fator organizacional importante e útil.

Para quem não sabe, uma interface é uma definição da estrutura de uma classe. Ela define todos os métodos que ela terá.

```

public interface IRepository
{
    // Métodos genéricos
    void Add<T>(T entity) where T : class;
    void Update<T>(T entity) where T : class;
    void Delete<T>(T entity) where T : class;
    Task<bool> SaveChangesAsync();

    //Métodos GET, que não são genéricos, pois possuem algo específico,
    no caso, Aluno!
    Task<Aluno[]> GetAllAlunosAsync();
    Task<Aluno> GetAllAlunosAsyncByRa(string RA);
}

```

Repository.cs é uma classe escrita na linguagem de programação C# que representa o repositório a ser utilizado em si. É nela que escreveremos o código de todos os métodos da interface. Mostraremos para que ela serve mais para frente na seção do AlunoController.cs.

```

public class Repository : IRepository
{
    public EscolaContext Context { get; }

    public Repository(EscolaContext context)
    {
        this.Context = context;
    }

    public void Add<T>(T entity) where T : class
    {
        this.Context.Add(entity);
    }

    public void Delete<T>(T entity) where T : class
    {
        this.Context.Remove(entity);
    }

    public async Task<bool> SaveChangesAsync()
    {

```

```

        // Como é tipo Task vai gerar thread, então vamos definir o
método como assíncrono (async)
        // Por ser assíncrono, o return deve esperar (await) se tem
alguma coisa para salvar no BD
        // Ainda verifica se fez alguma alteração no BD, se for maior
que 0 retorna true ou false
        return (await this.Context.SaveChangesAsync() > 0);
    }

    public void Update<T>(T entity) where T : class
    {
        this.Context.Update(entity);
    }

    public async Task<Aluno[]> GetAllAlunosAsync()
    {
        //Retornar para uma query qualquer do tipo Aluno
        IQueryable<Aluno> consultaAlunos = this.Context.Alunos;
        consultaAlunos = consultaAlunos.OrderBy(a => a.RA);
        // aqui efetivamente ocorre o SELECT no BD
        return await consultaAlunos.ToArrayAsync();
    }

    public async Task<Aluno> GetAllAlunosAsyncByRa(string RA)
    {
        //Retornar para uma query qualquer do tipo Aluno
        IQueryable<Aluno> consultaAlunos = this.Context.Alunos;
        consultaAlunos = consultaAlunos.OrderBy(a => a.RA).Where(aluno
=> aluno.RA == RA);
        // aqui efetivamente ocorre o SELECT no BD
        return await consultaAlunos.FirstOrDefaultAsync();
    }
}

```

Podemos observar, primeiramente, a existência de métodos assíncronos. Mas, o que eles são e o que definem? Basicamente, eles definem que o código será executado simultaneamente com outros. E, para impedir que o programa se enrole e se perca no meio do caminho, utilizamos o await, que faz com que o programa espere o código terminar de executar.

Async == Assíncrono == Execução Simultânea

Também podemos observar a presença do EscolaContext, que será o contexto utilizado no nosso repositório. Ele receberá o nome Context, como

escrito no construtor. Aproveitando, os métodos Add, Delete e Update serão escritos utilizando-se somente desse tão falado contexto. Já os métodos Get são um pouco mais complexos e irão requerer uma explicação mais aprofundada. Continue conosco no próximo parágrafo!

Certo, chegou a hora desses Gets. Basicamente, criamos uma instância da classe IQueryable de alunos e fazemos ela receber todos os nossos alunos presentes no contexto, chamando a propriedade Alunos, e depois ordenamos esses alunos em ordem de RA, retornando-os em forma de array (vetor). Antes de retornar esses alunos, também podemos fazer uma busca mais precisa, como somente retornar aqueles com RA 19192, ou até 19168, utilizando o .Where.

E, por fim, o método SaveChangesAsync é o mais importante de TODOS, pois os métodos que alteram informações do banco dependem dele. É o método que salva essas informações. É como entrar em um arquivo, alterar coisas, e não salvar, perdendo tudo no final. É bom estarmos atentos que este método não é um simples boolean não, é algo MAIOR, ele é uma **Task**, que entra em loop até as coisas se salvarem no BD, e então ela retorna true (ou 1).

AlunoController.cs

AlunoController.cs é uma classe escrita na linguagem de programação C# que representa o controlador de todos os alunos. Ela tem por função realizar todas as operações necessárias chamadas a partir do que chamamos de rota, que, basicamente, é o que está na URL de seu navegador, utilizando o repositório.

```
[Route("api/[controller]")]
[ApiController]
public class AlunoController : Controller
{
    public IRepository Repo { get; }
    public AlunoController(IRepository repo)
    {
        this.Repo = repo;
        //construtor
    }

    [HttpGet]
    public async Task<IActionResult> Get()
    {
        var result = await this.Repo.GetAllAlunosAsync();
        return Ok(result);
    }
}
```

```

[HttpGet("{AlunoRA}")]
public async Task<IActionResult> Get(string AlunoRA)
{
    try
    {
        var result = await
this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
        return Ok(result);
    }
    catch
    {
        return
this.StatusCode(StatusCodes.Status500InternalServerError, "Falha no
acesso ao banco de dados.");
    }
}

[HttpPost]
public async Task<IActionResult> post(Aluno model)
{
    try
    {
        this.Repo.Add(model);

        if (await this.Repo.SaveChangesAsync())
        {
            //return Ok();
            return Created($"/api/aluno/{model.Id}", model);
        }
    }
    catch
    {
        return
this.StatusCode(StatusCodes.Status500InternalServerError, "Falha no
acesso ao banco de dados.");
    }

    return BadRequest();
}

```



```

[HttpPut("{AlunoRA}")]
public async Task<IActionResult> put(string AlunoRA, Aluno model)
{
    try
    {
        this.Repo.Update(model);

        if (await this.Repo.SaveChangesAsync())
        {
            //return Ok();
            //pegar o aluno novamente, agora alterado para devolver
pela rota abaixo

            var aluno = await
this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
            return Created($"api/aluno/{model.RA}", aluno);
        }
    }
    catch
    {
        return
this.StatusCode(StatusCodes.Status500InternalServerError, "Falha no
acesso ao banco de dados.");
    }

    return BadRequest();
}

[HttpDelete("{AlunoRA}")]
public async Task<IActionResult> delete(string AlunoRA)
{
    try
    {
        //verifica se existe aluno a ser excluído
        var aluno = await this.Repo.GetAllAlunosAsyncByRa(AlunoRA);
        if (aluno == null)
            return NotFound(); //método do EF
        this.Repo.Delete(aluno);

        if (await this.Repo.SaveChangesAsync())
        {
            return Ok();
        }
    }
    catch
    {
        return BadRequest();
    }
}

```

```

    }
}
catch
{
    return
this.StatusCode(StatusCode.Status500InternalServerError, "Falha no
acesso ao banco de dados.");
}
return BadRequest();
}
}

```

Podemos observar que, antes de todos os métodos assíncronos, possuímos uma definição mais precisa da rota -[Route("api/[controller]")]- e de sua função, no caso um **controller**. Observe mais atentamente os métodos, agora, para entender do que estamos falando, e preste bastante atenção nos "{AlunoRA}", pois são eles que fazem o método receber o RA passado na URL como parâmetro.

Certo, hora de aprofundarmos ainda mais nossos conhecimentos: todos os métodos retornam um valor, que pode ser o que foi criado, ou apenas uma simples confirmação ou mensagem de sucesso. Também podemos ver que os métodos **Get** sempre criam uma instância que receberá alunos a partir do repositório e a retorna, que o método **Post** adiciona o aluno pelo repositório, salva as alterações e retorna o que foi criado, que o método **Put** atualiza o aluno pelo repositório, salva as alterações e retorna o que foi atualizado, e que o método **Delete** verifica se existe um aluno com o RA passado e então, se existir, ele deleta esse aluno pelo repositório e salva as alterações, mas, se não existir, ele retorna um erro (não encontrado).

Como uma observação final, é importante observarmos também que, para todos os **controllers**, temos uma regra de nomenclatura: sempre precisaremos ter a palavra Controller, com "C" maiúsculo, no final. Por exemplo, UmController, ProfessorController, MatematicaController, sendo a primeira palavra uma de sua escolha. Precisamos escrever desta forma porque, na hora de escrevermos a URL, teremos que utilizar a primeira palavra do nome do nosso controlador, como no exemplo <https://localhost:5001/aluno>.



Fontes

- <https://docs.microsoft.com/pt-br/aspnet/core/web-api/action-return-types?view=aspnetcore-3.1>;
- Apostila ASP.NET Core 3.1 com Entity Framework SQL Server da professora Patrícia.