

Laboratório de Desenvolvimento de Software

Célio Carvalho
cdf@estg.ipp.pt

P. PORTO

CONTEXTUALIZAÇÃO (1/2)

- Uma WebAPI é um conjunto de métodos públicos que permite que várias aplicações possam comunicar entre si de forma transparente e simples via internet. Os métodos públicos são, por vezes, chamados de *WebMethods*, porque têm funcionalidade similar à execução de métodos mas através da rede pública.
- Os *WebMethods* podem ser vistos como forma de expor funcionalidades e/ou dados de um sistema, utilizando o protocolo HTTP ou HTTPS. Isto significa que, o consumo deste métodos, é feito através de *Uniform Resource Locators* (URL), usando métodos HTTP como o GET, POST, PUT e DELETE.
- As informações normalmente transmitidas no *body* dos *requests* ou *responses* podem assumir vários formatos. Os mais comuns serão JavaScript Object Notation (JSON) e eXtensible Markup Language (XML). O mais utilizado será o JSON, talvez por ser simples de usar, e o *overhead* dos metadados ser inferior ao XML.
- As WebAPI são ***stateless***, o que significa que cada *request* é independente e não mantém informações de estado de *requests* anteriores. Isto quer dizer que, os dados de contexto necessários ao processamento de um *request*, têm que ser fornecidos no próprio *request* como se este fosse o primeiro a ser efetuado.
- As WebAPI contemplam normalmente medidas de segurança para garantir autenticação e autorização.

CONTEXTUALIZAÇÃO (2/2)

- As WebAPI utilizam-se em vários contextos nos dias de hoje:
 - **Integração de aplicações** – A comunicação entre programas torna-se mais simples e transparente utilizando WebAPI. Por exemplo, uma aplicação pode aceder a outra para obter a morada de um cliente já existente nessa outra aplicação;
 - **Serviços Web** – Existem serviços web que processam ou fornecem dados para serem consumidos noutras aplicações. Por exemplo, existem serviços que convertem um valor recebido numa moeda (e.g. euros) noutra moeda (dólares).
 - **Acesso a Dados** – São serviços que permitem aceder a informação existente. Por exemplo, informação meteorológica, cotações da bolsa, etc.
 - **Automação** – As WebAPI podem ser utilizados como *triggers* enviados para outro sistema. Por exemplo, a criação de uma encomenda, pode despoletar uma ordem de produção noutra aplicação.
 - **Internet of Things (IoT)** – As WebAPI podem ser consumidas pelos dispositivos IoT para comunicar telemetria. Por exemplo, um sensor de humidade pode comunicar a humidade do solo periodicamente, consumindo um WebMethod para o efeito.
- Nos dias de hoje, a utilização das WebAPI é generalizada. A sua utilização permite que se consiga efetuar comunicações entre sistemas e/ou aplicações de forma padronizada.

REST API | Contextualização

- Uma *Representational State Transfer Application Programming Interface* (REST API) resume um conjunto de regras e convenções a considerar no desenvolvimento de serviços de internet.
- Cada recurso é identificado por uma URL única. Um recurso pode ser um serviço, objeto, conjunto de objetos, etc. Cada recurso é transferido de e para o servidor usando um formato representativo (e.g. JSON).
- As ações dos recursos são mapeados para os métodos HTTP:
 - ✓ **GET**, para obter recursos;
 - ✓ **POST**, para criar um recurso;
 - ✓ **PUT**, para atualizar um recurso; e
 - ✓ **DELETE**, para eliminar um recurso.
- Os *requests* à API devem sempre conter toda a informação necessária ao processamento, porque não existe informação de contexto de *requests* anteriores (não há o conceito de sessão).
- A separação clara entre cliente e servidor, permite que ambos os sistemas possam evoluir de forma independente ou quase independente.

REST API | Exemplo

- A imagem abaixo apresenta uma REST API Exemplo

| API | Description | Request body | Response body |
|---|-------------------------|--------------|----------------------|
| <code>GET /api/todoitems</code> | Get all to-do items | None | Array of to-do items |
| <code>GET /api/todoitems/{id}</code> | Get an item by ID | None | To-do item |
| <code>POST /api/todoitems</code> | Add a new item | To-do item | To-do item |
| <code>PUT /api/todoitems/{id}</code> | Update an existing item | To-do item | None |
| <code>DELETE /api/todoitems/{id}</code> | Delete an item | None | None |

| API | Descrição | Request Body | Response Body |
|------------------------------------|---------------|---------------|-------------------------|
| <code>GET /api/people</code> | All People | Nenhum | Array de Objetos People |
| <code>GET /api/people/{id}</code> | Single People | Nenhum | Objeto People |
| <code>POST /api/people</code> | Create People | Objeto People | Objeto People |
| <code>PUT /api/todo/{id}</code> | Update People | Objeto People | Nenhum |
| <code>DELETE /api/todo/{id}</code> | Delete People | Nenhum | Nenhum |



.NET CORE | Contextualização (1/2)

- O ASP.NET Core é uma estrutura de código aberto de alto desempenho e multiplataforma para a criação de aplicações modernas, baseadas em *cloud computing* e ligadas à Internet.
- Com o ASP.NET Core, é possível: criar aplicações e serviços web, aplicações IoT, *backends* para dispositivos móveis e outros; desenvolver no Windows, macOS e Linux; fazer *deploy* na *cloud* ou localmente; etc.
- O .NET Core, através do ASP.Net, suporta a criação de WebAPI de forma simples. Tratando-se de código .NET, podem utilizar-se bibliotecas adicionais, para acesso a ficheiros, bases de dados, outros serviços web, etc.
- O ASP.Net Core é flexível e suporta várias formas de roteamento, autenticação (e.g. *tokens*, *cookies*), serialização (e.g. JSON, XML), etc. As WebAPI desenvolvidas em .NET Core podem ser alojadas em vários SO (e.g. Windows, Linux, Azure).
- Existem duas formas de criar WebAPI em .NET:
 - ✓ **Controller-based WebAPI**, que utiliza classes que derivam da classe ControllerBase.
 - ✓ **Minimal API**, define apenas *endpoints* com *handlers* lógicos sem o formalismo de uma classe base.
- Neste documento será utilizada a abordagem **Controller-based**.



.NET CORE | Contextualização (2/2)

- O ASP.NET Core:
 - ✓ Dispõe de tecnologia para consumir API existentes e desenvolver novas API;
 - ✓ O desenvolvimento pode ser suportado em testes automáticos;
 - ✓ Permite desenvolver páginas web de forma rápida através da tecnologia Razor Pages;
 - ✓ Como é multiplataforma, pode ser desenvolvido e alojado em vários SO;
 - ✓ Baseado em código aberto (comunidade ativa);
 - ✓ A configuração por defeito torna simples o *deployment* na *cloud*;
 - ✓ Contém *Dependency Injection*;
 - ✓ O pipeline para processamento de *requests* HTTP é simples e rápido;
 - ✓ As aplicações ASP.Net podem ser alojadas em vários serviços Web (e.g. IIS, Nginx (e.g. Docker));
 - ✓ Possibilidade de executar as aplicações em modo *auto-host* (i.e. executa no seu próprio processo);
 - ✓ Etc.
- Em resumo, o ASP.Net permite desenvolver aplicações Web modernas e rápidas. Podem utilizar-se padrões de desenvolvimento de software como MVC.

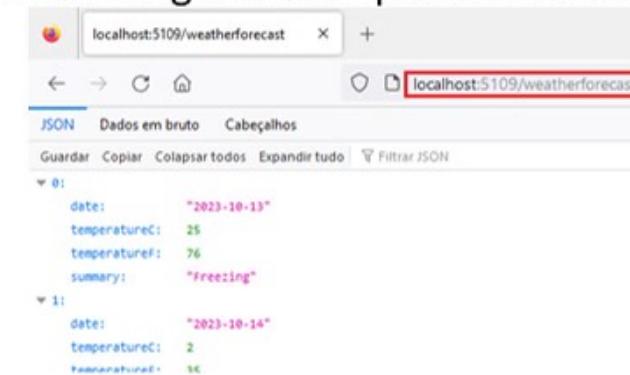
ASP.NET CORE | Command Line Interface (CLI)

- Os projetos ASP.Net podem ser manipulados através do Microsoft Visual Studio (VS) (ambiente gráfico), ou através da linha de comandos, através do *Command Line Interface (CLI)*. Neste documento será utilizado o VS, mas todas as operações de criação, refreshamento de dependências, compilação, testes, etc. podem ser executadas a partir da linha de comandos. As imagens abaixo apresentam alguns exemplos de utilização.

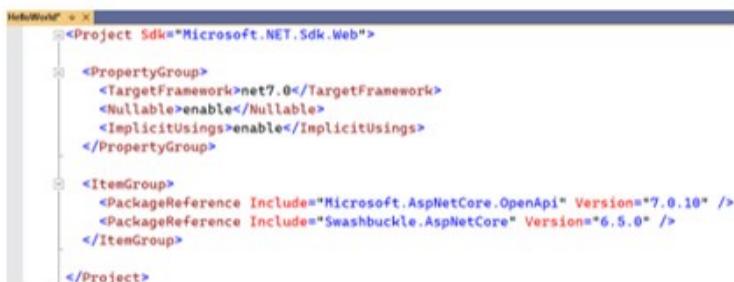
```
D:\clidemo\dotnet new webapi -o HelloWorld
The template "ASP.NET Core Web API" was created successfully.

Processing post-creation actions...
Restoring D:\clidemo\HelloWorld\HelloWorld.csproj
  Determining projects to restore...
  Restored D:\clidemo\HelloWorld\HelloWorld.csproj
Restore succeeded.

D:\clidemo\HelloWorld\dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5109
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\clidemo\HelloWorld
```



- As dependências do projeto podem ser personalizadas no ficheiro csproj.



CONTROLLER BASED | Demonstração (1/14)

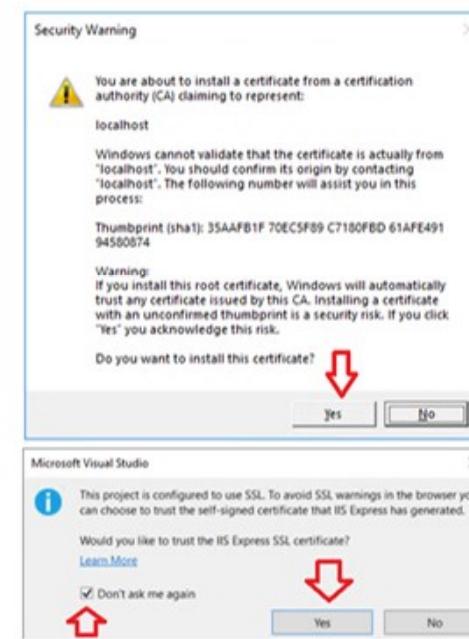
- Nesta demonstração vai ser criada uma pequena REST API utilizando a tecnologia **ASP.Net**. A ideia passa por criar um Controller que permita as operações *Create, Read, Update & Delete* (CRUD) básicas para lidar com os dados de uma entidade na base de dados.
- Nesta demonstração vai utilizar-se o *Entity Framework* (EF) para aceder a uma base de dados SQL Server. Através do EF será criado o *schema* da Base de Dados (BD) SchoolDB. As operações CRUD serão para gerir os dados da tabela Students.
- Crie um projeto **ASP.Net Core Web API** no Visual Studio. Note que é possível utilizar o *Command Line Interface* (CLI) do .NET para fazer esta criação. No entanto, nesta demonstração, será utilizado o Visual Studio para criar e desenvolver a solução.



CONTROLLER BASED | Demonstração (2/14)

- O Projeto adiciona automaticamente o *package* do *Swagger* para facilitar o desenvolvimento e documentação da solução.
- Sugere-se que utilize (também) o *Postman* (ou equivalente). A criação de *requests* pré-definidos facilitará os processos de teste posteriores. Nestas ferramentas é possível guardar o conteúdo dos *requests*, permitindo serem reexecutados posteriormente de forma rápida.
- Após terminar a criação da aplicação, pode executá-la de imediato. Apesar de não ter ainda desenvolvido qualquer linha de código, a aplicação de base já é funcional com um *Controller* disponibilizado por defeito durante a criação.
- Na primeira execução da aplicação, vai ser apresentada uma mensagem para confiar no certificado digital, por causa da utilização do SSL (utilização do HTTPS).

continua no slide seguinte...



The screenshot shows the Visual Studio interface. In the Solution Explorer, a red arrow points to the 'Packages' node under 'School.WebAPI', which contains two entries: 'Microsoft.AspNetCore.OpenApi (7.0.10)' and 'Swashbuckle.AspNetCore (6.5.0)'. In the 'Program.cs' code editor tab, another red arrow points to the 'appsettings.json' file. The code in 'Program.cs' is as follows:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI
// at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```



CONTROLLER BASED | Demonstração (3/14)

- Neste projeto vai ser utilizado o EntityFramework (EF) para persistir os dados numa BD SQL Server. Por esse motivo, a primeira tarefa no desenvolvimento do projeto, passa por incluir os *packages* NuGet seguintes:
 - Microsoft.EntityFrameworkCore.Design
 - Microsoft.EntityFrameworkCore.SqlServer
 - Microsoft.EntityFrameworkCore.Tools
- Na anatomia de um projeto WebAPI é frequente ver vários tipos de classes, das quais se destacam as classes de dados (*Models*) e as classes de serviço (*Controllers*) que processam os pedidos (*requests*) e enviam as respostas (*responses*) HTTP ou HTTPS.
- Analise a solução criada automaticamente pelo Visual Studio e localize o *Model* WeatherForecast e o *Controller* WeatherForecastController.
- Depois de entender o seu funcionamento, apague ambas as classes. Nesta demonstração será criada um novo *Model* e um novo *Controller* com *WebMethods* capazes de responder às operações CRUD de uma entidade.

continua no slide seguinte...

The screenshot shows the NuGet Package Manager interface with the following details:

- Installed** tab is selected.
- Search (Ctrl+L)**: Microsoft.AspNetCore.OpenApi by Microsoft
- Microsoft.AspNetCore.OpenApi** by Microsoft: Provides APIs for annotating route handler endpoints in ASP.NET Core with OpenAPI annotations.
- Microsoft.EntityFrameworkCore.Design** by Microsoft: Shared design-time components for Entity Framework Core tools.
- Microsoft.EntityFrameworkCore.SqlServer** by Microsoft: Microsoft SQL Server database provider for Entity Framework Core.
- Microsoft.EntityFrameworkCore.Tools** by Microsoft: Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.
- Swashbuckle.AspNetCore** by Swashbuckle.AspNetCore: Swagger tools for documenting APIs built on ASP.NET Core
- Connected Services**, **Dependencies**, **Properties**, **Controllers**, **appsettings.json**, **Program.cs**, and **WeatherForecast.cs** are listed under the project structure, with **WeatherForecastController.cs**, **appsettings.json**, **Program.cs**, and **WeatherForecast.cs** highlighted with red boxes.



CONTROLLER BASED | Demonstração (4/14)

- Inicie-se pela criação do Model Student. Esta classe será utilizada pelo EF para modelar e suportar os dados provenientes e com destino à BD.
- Esta classe será utilizada também como *Data Transfer Object* (DTO). No entanto, num ambiente mais real, seria recomendada a utilização de DTO para transferir dados entre a camada aplicacional e a camada de apresentação (*backend* ↔ *frontend*).
- Vantagens da utilização dos DTO:
 - ✓ Separação dos *Models* de domínio dos objetos utilizados pelo *fontend* (utilizados para transferir dados);
 - ✓ Aumento de desempenho porque o volume de informação a passar na rede é menor (os DTO têm, habitualmente um *subset* das propriedades do objeto de domínio original);
 - ✓ Segurança, porque apenas são transferidos os dados necessários para a camada de apresentação;
 - ✓ Promove a evolução da API sem que implique a evolução (imediata) da camada de apresentação (a alteração de propriedades no *Model* de domínio poderá não implicar a alteração no DTO);
 - ✓ Facilita que os mesmos dados (*Model*) possam ser representados de várias formas diferentes mediante o que é necessário para a camada de apresentações (DTO);
 - ✓ Etc.

```
public class Student
{
    0 references
    public int Id { get; set; }

    0 references
    public string Name { get; set; } = null;

    0 references
    public string? Address { get; set; }

    0 references
    public int Age { get; set; }

    0 references
    public bool Approved { get; set; }
}
```

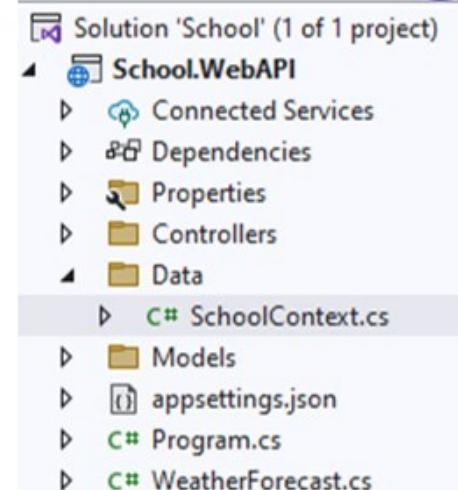
continua no slide seguinte...

CONTROLLER BASED | Demonstração (5/14)

- Após ter criado o *Model Student*, deverá agora criar a classe DbContext do EF que será responsável por ligar à BD SQL Server.
 - Crie a pasta Data; e
 - Adicione a classe SchoolContext dentro dessa pasta (ver imagem).

```
public class SchoolContext : DbContext
{
    0 references
    public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
    { }

    0 references
    public DbSet<Student> Students { get; set; }
}
```



- As *connection string* para ligação às bases de dados são, normalmente, colocadas num ficheiro de configurações. Neste caso, a solução já contém o ficheiro appsettings.json que serve, precisamente, para guardar estas informações.
 - Adicione ao ficheiro appsettings.json a *connection string* SchooldDb.

```
appsettings.json < X
Schema: https://json.schemastore.org/appsettings.json
1 {
2     "Logging": {
3         "LogLevel": {
4             "Default": "Information",
5             "Microsoft.AspNetCore": "Warning"
6         }
7     },
8     "AllowedHosts": "*",
9     "ConnectionStrings": {
10        "SchoolDB": "Data Source=.;Initial Catalog=SchoolDB;Integrated Security=True;TrustServerCertificate=True"
11    }
12 }
```

continua no slide seguinte...

CONTROLLER BASED | Demonstração (6/14)

- O ficheiro `program.cs` contém a lógica necessária para arranchar a WebAPI e disponibilizar os seus *endpoints* na rede para poderem ser consumidos. Assim, é neste ficheiro, que se adicionam os serviços e configurações necessárias à execução da WebAPI.
- Aceda ao ficheiro `program.cs`, e adicione o serviço ao arranque da aplicação. Utilize o método `UseSqlServer()` para especificar que se pretende ligar ao SQL Server, e também para especificar a *connection string* necessária a esta ligação.
- Note que, a anatomia do ficheiro `program.cs` tem evoluído ao longo das várias versões do ASP.Net. O código apresentado na imagem é válido para o **.NET 7.0**.
- Depois de alterar o `program.cs`, e antes de continuar com a implementação de código desta demonstração, valide que a solução compile sem erros.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<SchoolContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolDB"))
);

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

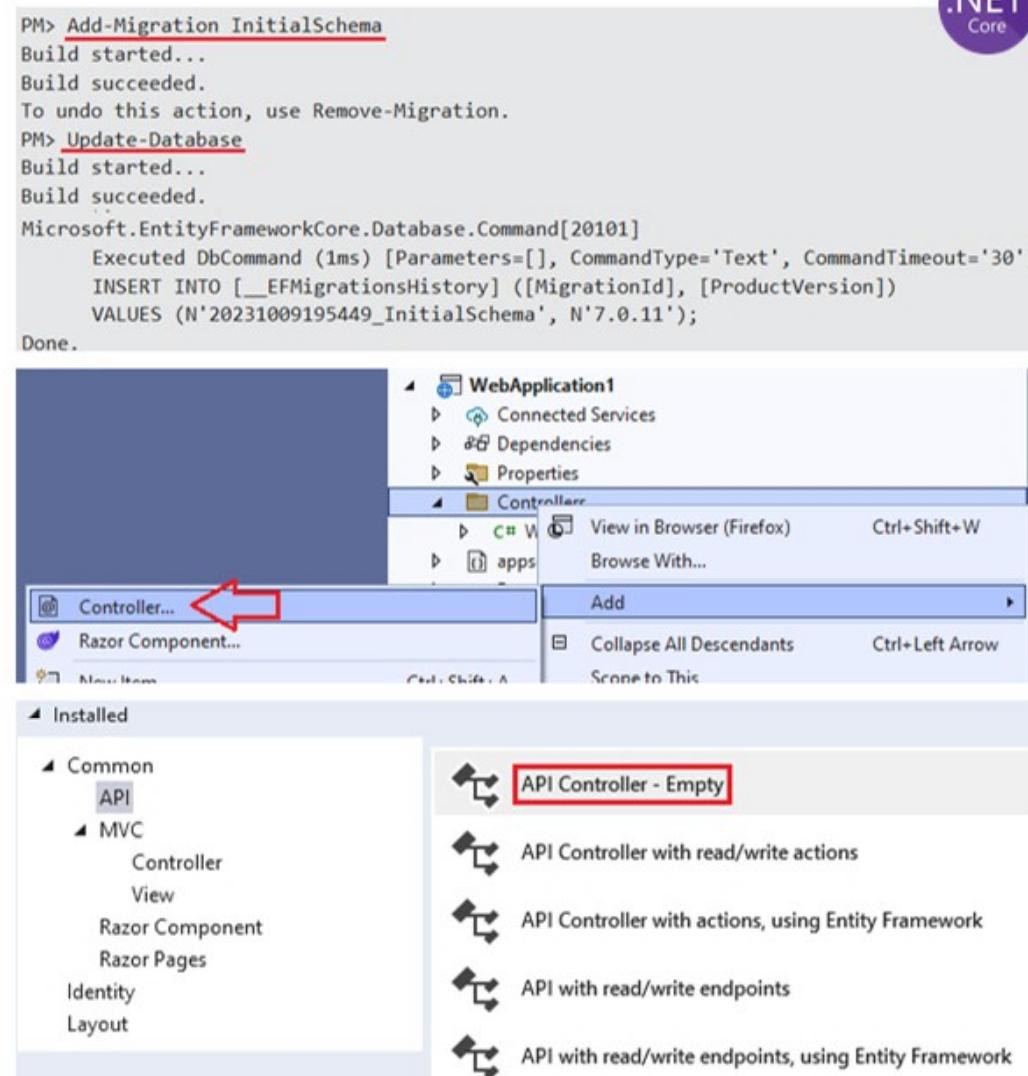
app.Run();
```

continua no slide seguinte...

CONTROLLER BASED | Demonstração (7/14)

- Segue-se a criação da base de dados e do *schema* respetivo. Para isso, crie uma EF *migration* e atualize a base de dados em conformidade.
 - AddMigration InitialSchema
 - Update-Database
- De seguida, chegou o momento de se criar o Controller que disponibilizará os métodos REST API de Students. Um Controller é uma classe que estende a classe ControllerBase.
- Adicione um novo Controller dentro da pasta Controllers, utilizando o botão direito do rato.
- No ecrã de seleção de *template*, selecione a opção correspondente a um Controller vazio.

continua no slide seguinte...



The screenshot shows the Visual Studio interface during the creation of a new controller. At the top, a command-line window displays the results of running migrations:

```
PM> Add-Migration InitialSchema
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20231009195449_InitialSchema', N'7.0.11');
Done.
```

In the center, the Solution Explorer shows a project named "WebApplication1" with a "Controllers" folder selected. A red arrow points to the "Controller..." item in the context menu that has been opened over the folder.

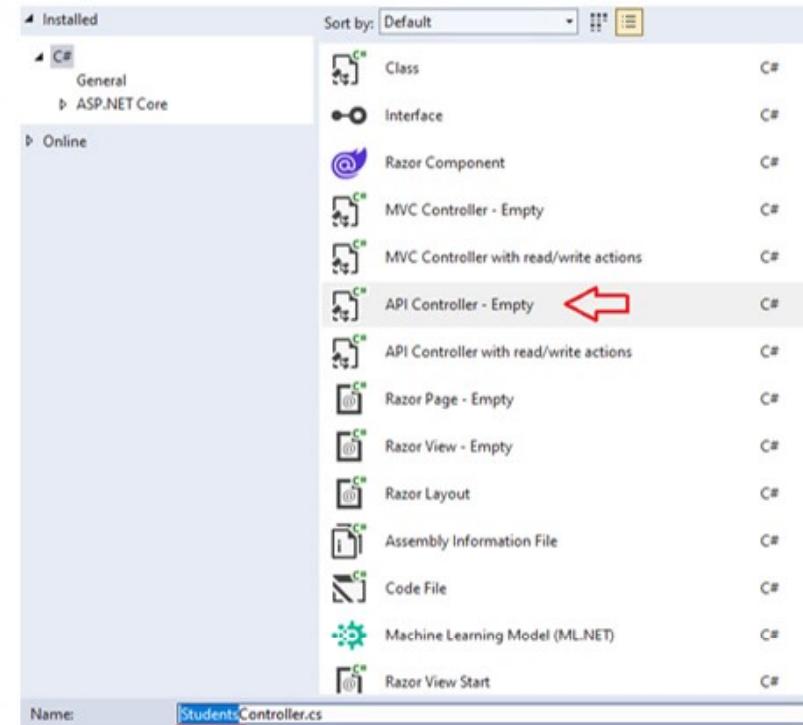
On the right, the "Add" context menu is open, showing various template options. One option, "API Controller - Empty", is highlighted with a red box.

Below the "Add" menu, a list of other available templates is shown:

- API Controller with read/write actions
- API Controller with actions, using Entity Framework
- API with read/write endpoints
- API with read/write endpoints, using Entity Framework

CONTROLLER BASED | Demonstração (8/14)

- Insira o nome do Controller mantendo o sufixo sugerido. Insira o nome `StudentsController.cs` como nome do ficheiro a criar.
- Note que o termo `Controller` deve existir no fim nome do ficheiro.
- Está a ser selecionado o template vazio, porque se pretende demonstrar a criação do Controller passo a passo. No entanto, no futuro, poderia utilizar outros *templates* para que sejam sugeridos *WebMethods* de base.
- De seguida serão criados os *endpoints* para as operações CRUD da REST API. Para testar o correto funcionamento dos métodos GET (leitura), insira diretamente na base de dados dois registo de Student (apenas para testes).



continua no slide seguinte...



CONTROLLER BASED | Demonstração (9/14)

- A classe Controller tem de ser decorada com duas Annotation:
 - ✓ **[ApiController]**, indica que a classe é um Controller da WebAPI; e
 - ✓ **[Route("...")]**, indica a URL ou template de URL em que a API deve responder.
- A Annotation **[Route()]** da imagem, indica que este Controller responderá na rota `api/students`. Notar que a parte inicial do endereço dependerá do servidor onde a WebAPI possa ser instalada.
- O acesso ao `DbContext` do EF estará disponível dentro do Controller através de *Dependency Injection (DI)*. O construtor da classe recebe uma referência ao `SchoolContext` que ficará disponível dentro da classe na variável `dbContext`. Para mais informações acerca de DI consulte o *link* abaixo.
 - ✓ <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class StudentsController : ControllerBase
{
    private readonly SchoolContext dbContext;
    0 references
    public StudentsController(SchoolContext schoolContext) => this.dbContext = schoolContext;
```

```
[Route("api/[controller]")]
[ApiController]
0 references
public class StudentsController : ControllerBase
{
    0 references
    ...
}
```

continua no slide seguinte...

CONTROLLER BASED | Demonstração (10/14)

- A primeiro método que será desenvolvido, é o que devolve todos os Student existentes na BD. Note que não é obrigatório que todas as operações CRUD sejam implementadas em todas as API.
- Notar a *Annotation* `[HttpGet]` no método `GetStudents()`. Esta configuração deve ser lida da seguinte forma: *quando chegar um request com o HttpMethod Get sem parâmetros de entrada, esse request deve ser atendido / respondido pelo método GetStudents()*. Resumindo, quando chegar um request GET para o endereço `api/students`, o comportamento especificado dentro do método `GetStudents()` é executado.
- Notar também o tipo de retorno `ActionResult<>` que contém os detalhes da resposta, i.e., *HttpCode* (e.g. `200OK`, `201CREATED`), *Headers* e *Body*. Para auxiliar na criação deste objeto, o ASP.Net disponibiliza métodos auxiliares (e.g. `Ok()`, `NotFound()`, `BadRequest()`).

```
// GET: api/Students
[HttpGet]
0 references
public ActionResult<IEnumerable<Student>> GetStudents()
{
    if (context.Students == null)
        return NotFound();

    return Ok(context.Students.ToList());
}
```

The screenshot shows a browser window with the title "Swagger UI". The address bar shows "localhost:44395/api/students". The main content area displays a JSON response with two entries:

```
[{"id": 1, "name": "C\'lio Carvalho", "address": "Avenida da Liberdade, Lisboa", "age": 20, "approved": true}, {"id": 3, "name": "Jo\u00e3o Jos\u00e9", "address": "Rua da Estrada", "age": 25, "approved": true}]
```

continua no slide seguinte...

CONTROLLER BASED | Demonstração (11/14)

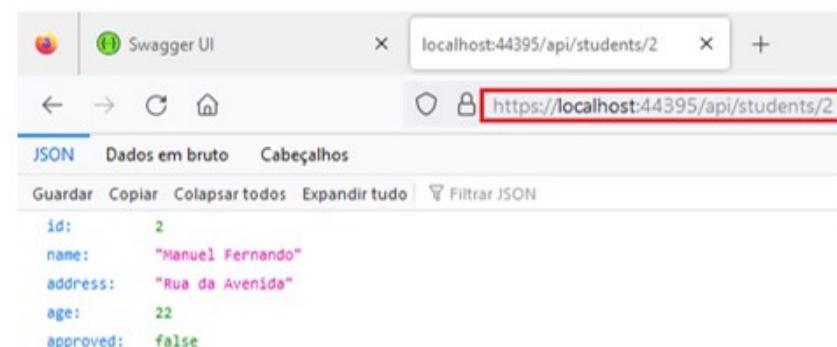
- O *endpoint* seguinte é similar ao anterior mas retorna apenas um *Student*. Neste caso, a *Annotation* `[HttpGet("{id}")]` informa que o parâmetro recebido na URL deverá ser mapeado para o argumento de entrada *id*. Por sua vez, o próprio método espera a receção de um argumento com o mesmo nome, do tipo inteiro.
- Ao pedir o consumo do método (e.g. Swagger, Postman, ou um qualquer browser), repare que o *id* é especificado na URL como um argumento. Quando o endpoint responde, passa o valor 2 para o argumento de entrada do método, e este processa o comportamento com base nesse valor.
- Para melhor entender o funcionamento do método, coloque um *breakpoint* na primeira linha do corpo do método, e execute a WebAPI em modo *debug*.

```
// GET: api/students/2
[HttpGet("{id}")]
public ActionResult<Student> GetStudent(int id)
{
    if (context.Students == null)
        return NotFound();

    var student = context.Students.SingleOrDefault(s => s.Id == id);

    if (student == null)
        return NotFound();

    return Ok(student);
}
```



continua no slide seguinte...

CONTROLLER BASED | Demonstração (12/14)

- De seguida apresenta-se o *endpoint* para a criação de um recurso, i.e., para a criação de um Student. Uma vez mais, note que a escolha do método Add() a ser executado aquando de um *request* com a rota api/students, é feita tendo por base do HttpMethod do *request*. Como é um POST o Controller encaminhará o *request* para este *endpoint*.
- Os dados do novo recurso são enviados no *body* do *request*. As imagens apresentadas são do Swagger. Note a utilização do método POST na parte superior da janela.
- Analise o resultado obtido, nomeadamente o *HttpCode* devolvido (201 CREATED), e a relação com o método que foi executado na instrução do return do método Add().
- Consulte a BD para validar a inserção do novo recurso (registo).

continua no slide seguinte...

| Id | Name | Address | Age | Approved |
|------|------------------|-------------------|------|----------|
| 1 | Célio Carvalho | Avenida da Lib... | 20 | True |
| 2 | Manuel Fernan... | Rua da Avenida | 22 | False |
| 3 | Maria José | Rua da Esquina | 24 | False |
| NULL | NULL | NULL | NULL | NULL |

```
// POST: api/students
[HttpPost]
1 reference
public ActionResult<Student> Add(Student student)
{
    // business rules validation
    // (...)

    context.Students.Add(student);
    context.SaveChanges();
    return CreatedAtAction(nameof(Add), new { id = student.Id }, student);
}
```

| POST | /api/Students |
|--|------------------|
| Parameters | Cancel |
| No parameters | Reset |
| Request body | application/json |
| <pre>{ "id": 0, "name": "Maria José", "address": "Rua da Esquina", "age": 24, "approved": false }</pre> | |
| 201 | Undocumented |
| Response body | |
| <pre>{ "id": 3, "name": "Maria José", "address": "Rua da Esquina", "age": 24, "approved": false }</pre> | |
| Download | |
| Response headers | |
| <pre>content-type: application/json; charset=utf-8 date: Mon, 09 Oct 2023 20:38:16 GMT location: https://localhost:44395/api/Students?id=3 server: Microsoft-IIS/10.0 x-firefox-spdy: h2 x-powered-by: ASP.NET</pre> | |
| Responses | |
| Execute | Clear |

CONTROLLER BASED | Demonstração (13/14)

- O PUT é o *HttpMethod* utilizado nas REST API para atualizar um recurso. Note a utilização da Annotation `[HttpPut("{id}")]` no método `Update()`. O *id* indicado é, também, mapeado para o *id* do argumento de entrada do método `Update()`.
- Por convenção, quando se envia um *request* para alterar os dados de um recurso, envia-se os novos dados no *body*, mas também se envia o *id* na URL do *request*. Este *id* tem que ser o mesmo do recurso indicado no *body*.
- Notar que o tipo de retorno do método `Update()` é o `IActionResult`. A *response* não tem *body* e apenas se precisa devolver o *HttpCode* `204NOCONTENT`.
- A instrução delineada a vermelho, muda o estado do objeto `Student` no EF para *Modified*. Se esta alteração não for feita, o EF não persistirá os dados na base de dados por considerar desnecessário.

continua no slide seguinte...

The screenshot illustrates a .NET Core Web API demonstration. On the left, a code editor shows a C# controller method for updating a student:

```
//PUT: api/students/{id}
[HttpPut("{id}")]
public IActionResult Update(int id, Student student)
{
    if (!student.Id.Equals(id))
        return BadRequest();

    // business rules validation
    // (...)

    context.Students.Entry(student).State = EntityState.Modified;
    context.SaveChanges();
    return NoContent();
}
```

On the right, a browser interface is shown for a PUT request to `/api/Students/{id}`. The URL parameter `id` is set to 3. The Request body contains the following JSON:

```
{
    "id": 3,
    "name": "João José",
    "address": "Rua da Estrada",
    "age": 25,
    "approved": true
}
```

The browser response shows a `204 Undocumented` status with the following headers:

- date: Mon, 09 Oct 2023 20:48:46 GMT
- server: Microsoft-IIS/10.0
- x-firefox-spdy: h2
- x-powered-by: ASP.NET

Below the browser interface, a table displays student data:

| ID | Name | Address | Age | Approved |
|----|------------------|-------------------|-----|----------|
| 1 | Célio Carvalho | Avenida da Lib... | 20 | True |
| 2 | Manuel Fernan... | Rua da Avenida | 22 | False |
| 3 | João José | Rua da Estrada | 25 | True |

At the bottom, there are `Execute` and `Clear` buttons.

CONTROLLER BASED | Demonstração (14/14)

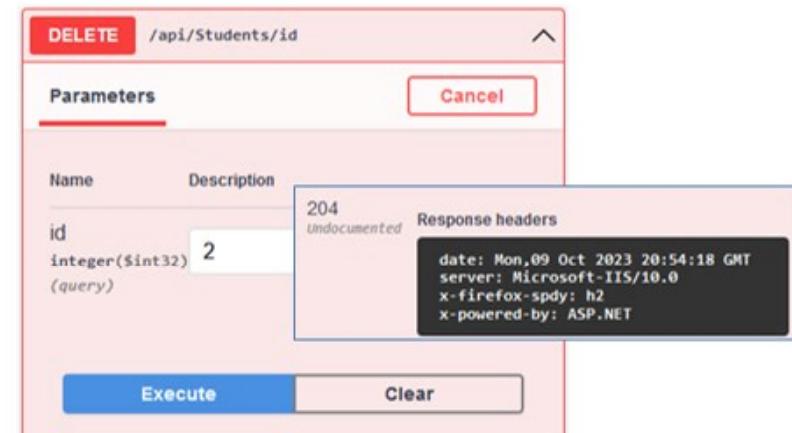
- Resta demonstrar a utilização do *HttpMethod DELETE* para remover um recurso no servidor. Note a utilização do método `[HttpDelete("{id}")]`. O *id* indicado é, mais uma vez, mapeado para o *id* do argumento de entrada do método `Delete()`.
- A utilização do método `DELETE` não necessita de *body*. O recurso a eliminar é identificado através do argumento enviado na URL do *request*.
- Neste caso, utiliza-se novamente o `IActionResult` como tipo de retorno. A *response* também não tem *body* e apenas se precisa devolver o *HttpCode 204NOCONTENT*.
- Teste os vários *endpoints*, e analise os impactos na BD para aferir o seu correto funcionamento.

```
//DELETE: api/students/2
[HttpDelete("{id}")]
0 references
public IActionResult Delete(int id)
{
    if (context.Students == null)
        return NotFound();

    var student = context.Students.SingleOrDefault(s => s.Id == id);

    if (student == null)
        return NotFound();

    context.Students.Remove(student);
    context.SaveChanges();
    return NoContent();
}
```



| Id | Name | Address | Age | Approved |
|------|----------------|-------------------|------|----------|
| 1 | Célio Carvalho | Avenida da Lib... | 20 | True |
| 3 | João José | Rua da Estrada | 25 | True |
| NULL | NULL | NULL | NULL | NULL |



LINKS ÚTEIS

- ***Parameter Binding in ASP.NET Web***
 - <https://learn.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/parameter-binding-in-aspnet-web-api>
- ***Create web APIs with ASP.NET Core***
 - <https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-7.0>
- ***Tutorial: Create a web API with ASP.NET Core***
 - <https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio>
- ***Build a RESTful Web API with ASP.NET Core 6***
 - <https://medium.com/net-core/build-a-restful-web-api-with-asp-net-core-6-30747197e229>

P. PORTO