

# Laboratório de Desenvolvimento de Software

Célio Carvalho  
[cdf@estg.ipp.pt](mailto:cdf@estg.ipp.pt)

P. PORTO



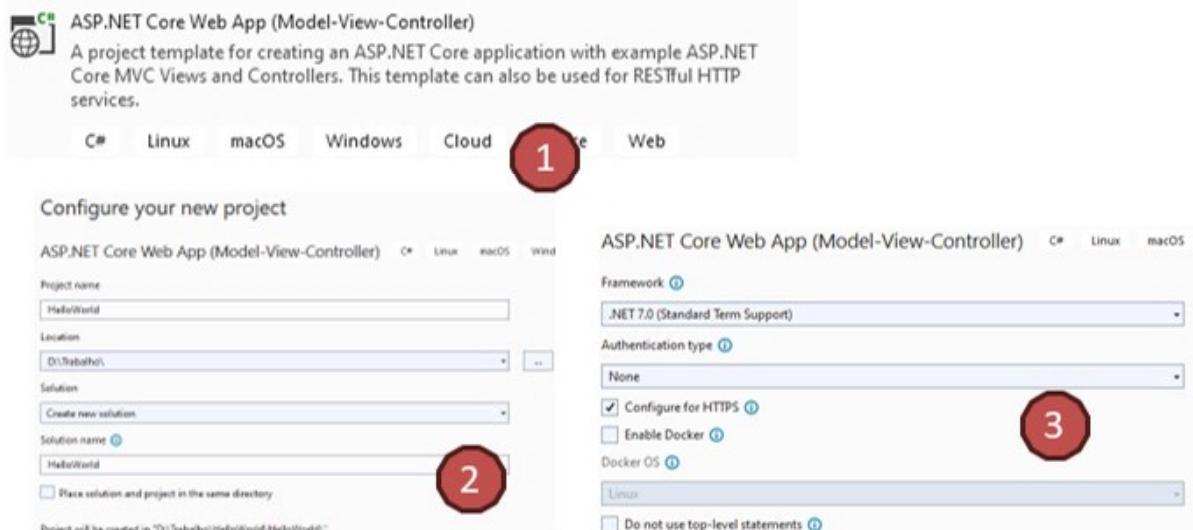
## CONTEXTUALIZAÇÃO

- O Razor é um mecanismo baseado em Views do ASP.Net Core que permite criar interfaces web de forma rápida e dinâmica.
- Uma das características do Razor é permitir que instruções de execução C# (ou outra linguagens .NET) possam ser misturadas (embebidas) com *markup* HTML.
- Características principais:
  - ✓ A sintaxe simples, como se baseia em HTML embebido em C#, é fácil de ler e entender;
  - ✓ A integração com o código facilita a geração dinâmica de conteúdo e integração com dados;
  - ✓ O Razor permite o desenvolvimento de componentes reutilizáveis, promovendo o desenvolvimento mais rápido e de mais fácil manutenção;
  - ✓ As Tag Helpers facilitam a fusão de código com HTML;
  - ✓ O programador pode adicionar Tag Helpers para personalizar o processo de desenvolvimento;
  - ✓ Etc.

continua no slide seguinte...

## WEB | Demonstração 1 (1/5)

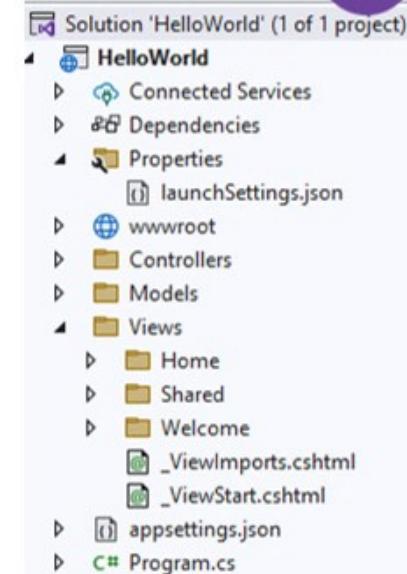
- Nesta pequena demonstração, pretende-se introduzir a tecnologia **Razor**, e apresentar a anatomia de uma solução. Nesta demonstração não será utilizado o *Command Line Interface* (CLI) do .NET. No entanto, caso prefira, poderá optar por essa via.
  - ✓ Utilize o Microsoft Visual Studio (VS) para criar uma aplicação ASP.NET Core MVC.
  - ✓ Chame à solução e ao projeto HelloWorld.
  - ✓ Selecione a .NET 7.0 e assuma os valores sugeridos nos restantes parâmetros.
  - ✓ Posteriormente, execute a aplicação para ver em funcionamento o conteúdo que é sugerido de base.
  - ✓ Na mensagem que pede para confiar no certificado digital, responda afirmativamente.



continua no slide seguinte...

## WEB | Demonstração 1 (2/5)

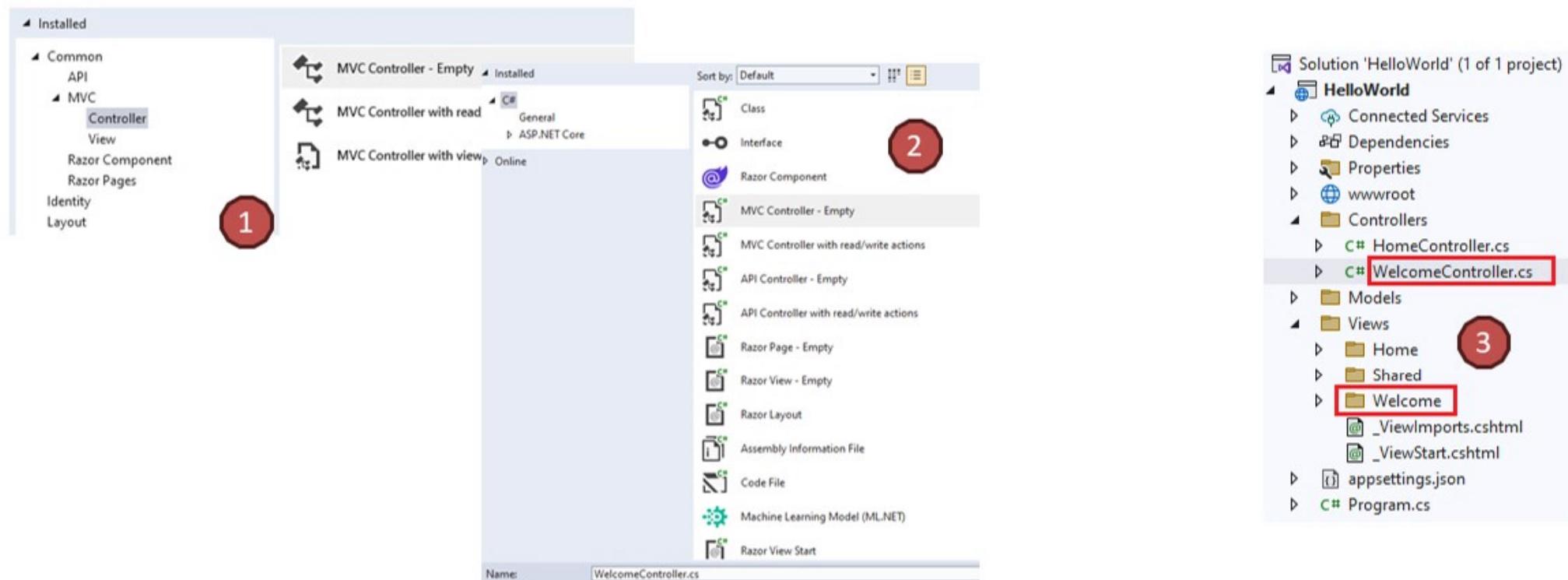
- Descreve-se de seguida a anatomia de um projeto Razor:
  - ✓ **program.cs**, responsável pelo arranque da aplicação, permite adicionar rotas na aplicação, etc.;
  - ✓ **appsettings.json**, ficheiro JSON onde se podem inserir parâmetros da aplicação (e.g. connectionStrings);
  - ✓ **wwwroot**, pasta onde se pode colocar todo o conteúdo estático (e.g. imagens, CSS, JS, etc.);
  - ✓ **Controllers**, é onde vão ficar os *controllers* da arquitetura MVC (e.g. sempre que a ação *index* seja chamada, haverá um redireccionamento para a *View* com o mesmo nome);
  - ✓ **Views**, contém os ficheiros \*.cshtml (ficheiros HTML com C#). Sempre que um *Controller* é criado, deve também existir uma pasta corresponde nas *Views*;
  - ✓ **Models**, contém as classes entidade (classes que contêm os dados);
  - ✓ **launchSettings.json**, contém os perfis de execução do projeto;
  - ✓ **\_ViewImports**, utilizado para incluir no projeto usings ou referencias ao nível global (visíveis em todas as views);
  - ✓ **\_ViewStart**, Define a View mestre (aquele que contém o @RenderBody()).



continua no slide seguinte...

## WEB | Demonstração 1 (3/5)

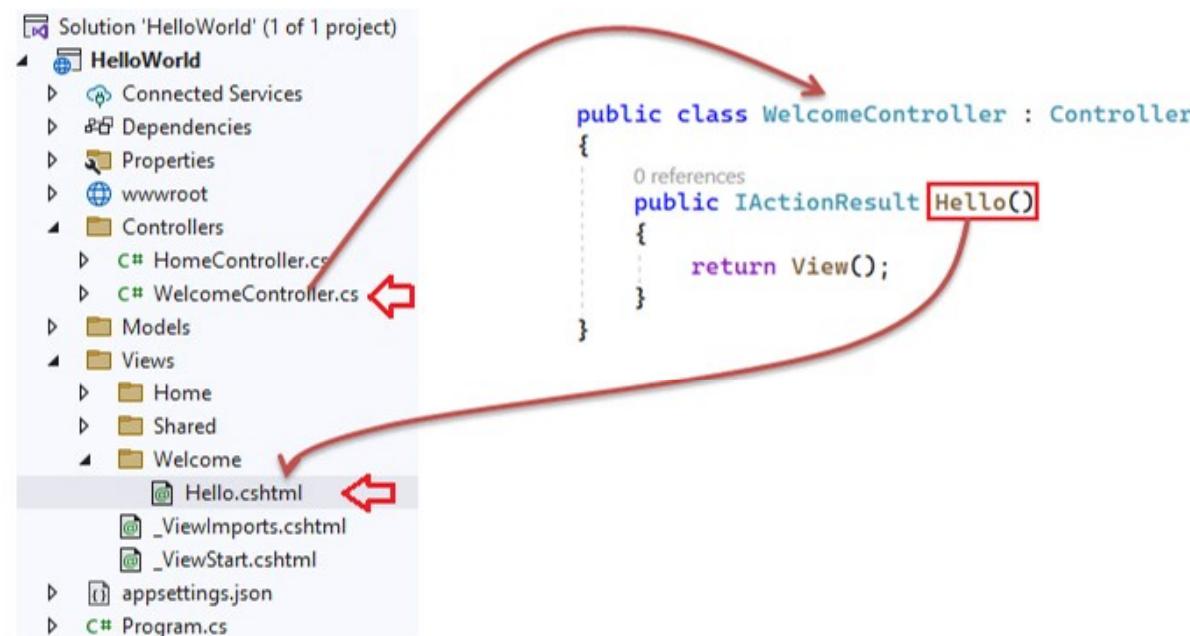
- Adicione um Controller com o nome Welcome. Tem que existir uma pasta com o mesmo nome dentro da pasta Views. Note que deverá manter o sufixo Controller.cs no final do nome do Controller.



continua no slide seguinte...

## WEB | Demonstração 1 (4/5)

- Personalize o conteúdo do controller criado, adicionando o método Hello(). A View( ) correspondente a este método (*action*) deve ter o mesmo nome deste método.
  - Resumindo, tem que haver uma pasta com o nome do controller dentro da pasta Views, e tem que haver uma View dentro da pasta Views com o mesmo nome do método.



continua no slide seguinte...

## WEB | Demonstração 1 (5/5)

- Edite o conteúdo da View com o código apresentado na imagem. Posteriormente execute a aplicação e navegue no *browser* usando o nome do novo controller (cf. Welcome) e da nova action (Hello).



continua no slide seguinte...



## WEB | Exemplos breves

- O Razor é uma *framework* que permite embeber código tipicamente implementado no lado do servidor diretamente nas páginas *web*;
- Na tecnologia *WebPages*, existe uma separação entre o que é HTML e código executado do lado do *browser (front-end)* e o que é código executado do lado do servidor (*backend*) que permite a comunicação dos utilizadores a partir do *frontend*.

```
<h1>Razor demo 1</h1>
<p>Data e hora atual: @DateTime.Now.ToString()</p>
```

```
<h1>Razor demo 2</h1>
<p>
@if (DateTime.Now.Day % 2 == 0)
{
    @DateTime.Now.Day
    <span>dia par.</span> @* ou @:estamos num dia par. *@
}
else
{
    @:hoje é dia ímpar      @* <span>hoje é dia ímpar</span> *@
}
</p>
```

```
<h1>Razor demo 4</h1>
<ul>
@{
    for (int i = 0; i < 5; i++)
    {
        <li>@($"Este é o item { i.ToString("0000") }!")</li>
    }
}
</ul>
```

```
<h1>Razor demo 3</h1>
<p>
 @{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Thursday)
        @($" { DateTime.Now } hoje é quinta-feira!")
    else
        @("hoje não é quinta-feira!")
 }
</p>
```

```
<h1>Razor demo 5</h1>
<ul>
@{
    var data = new DateTime(2040, 1, 1);

    for (int i = 0; i < 31; i++)
    {
        <li>
        ...
        @($"º dia { data.AddDays(i).ToString("yyyy-MM-dd") } , é { data.AddDays(i).DayOfWeek }")
        </li>
    }
}
</ul>
```

## WEB | Actions

- Os métodos implementados no Controller, que funcionam como *handlers* para execução de Views, são chamados de ações. O tipo de dados típico devolvido pelas ações são do tipo genérico `IActionResult` ou `ActionResult` (classe abstrata).
- Diferentes tipos de ações podem devolver diferentes tipos de `ActionResult`. O tipo específico de `ActionResult` depende do que a ação faz e como se deseja que a resposta seja formatada. O quadro abaixo resume os tipos mais utilizados de `ActionResult`.

<code>ViewResult</code>	Tipo específico para apresentar uma View no browser.
<code>JsonResult</code>	Tipo que devolve dados em formato JSON.
<code>ContentResult</code>	Permite devolver dados em formato personalizado (e.g. formato específico).
<code>FileResult</code>	Tipo que devolve uma resposta do tipo ficheiros (utilizado em downloads).
<code>RedirectToAction</code>	Redireciona o cliente para outra URL (usado, por exemplo, após terminar uma ação).
<code>BadRequestResult</code>	Devolve um <code>HttpCode 400 (Bad Request)</code> .
<code>NotFoundResult</code>	Devolve um <code>HttpCode 404 (Not Found)</code> .
<code>UnauthorizedResult</code>	Devolve um <code>HttpCode 401 (Unauthorized)</code> .

## WEB | Helper Methods

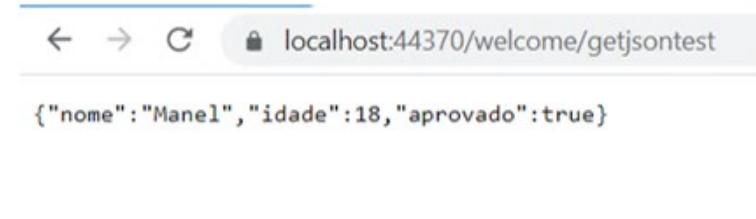
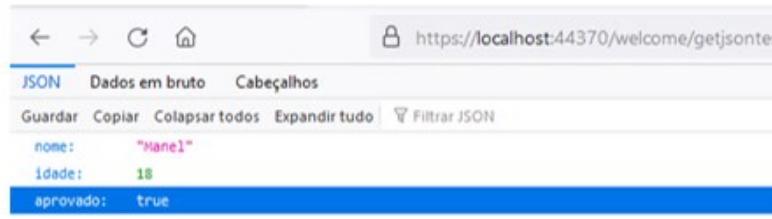
- O quadro resumo seguinte apresenta os *HelperMethods* que devolvem cada um dos *ActionResult*.

Action Result	Helper Method	Description
ViewResult	View	Renders a view as web page
PartialViewResult	PartialView	Renders a partial view, which defines a section of a view that can be rendered inside another view.
RedirectResult	Redirect	Redirects to another action method by using URL
RedirectToRouteResult	RedirectToAction or RedirectToRoute	Redirects to another action method
ContentResult	Content	Returns a user-defined content type
JsonResult	Json	Returns serialized json object
JavaScriptResult	JavaScript	Returns a script that can be executed on the client.
HttpStatusCodeResult	(None)	Returns a specific HTTP response code and description.
HttpUnauthorizedResult	(None)	Returns the result of an unauthorized HTTP request.
HttpNotFoundResult	HttpNotFound	Indicates the requested resource was not found.
FileResult	File	Returns binary output to write to the response.
FileContentResult	Controller.File(Byte[], String) or Controller.File(Byte[], String, String)	Sends the contents of a binary file to the response.
FilePathResult	Controller.File(String, String) or Controller.File(String, String, String)	Sends the contents of a file to the response.
FileStreamResult	Controller.File(Stream, String) or Controller.File(Stream, String, String)	Sends binary content to the response through a stream.
EmptyResult	(None)	Represents a return value that is used if the action method must return a <b>null</b> result (void).

## WEB | Helper Methods (exemplo)

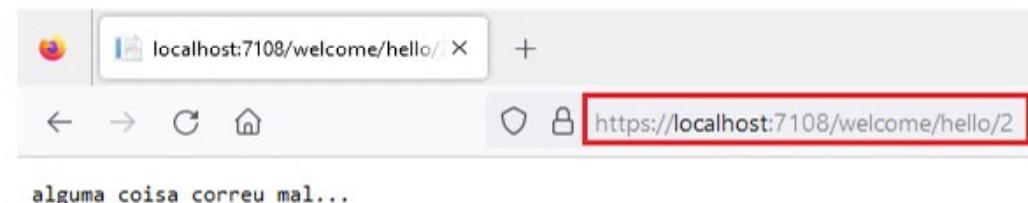
- Os imagens seguintes apresenta um exemplo simples de como devolver um JsonResult utilizando o ActionResult genérico a o método Json().

```
public ActionResult GetJsonTest()  
{  
    return Json(new {  
        nome = "Manel",  
        idade = 18,  
        aprovado = true  
    });  
}
```



- O exemplo seguinte devolve tipos específicos de ActionResult em função do processamento interno.

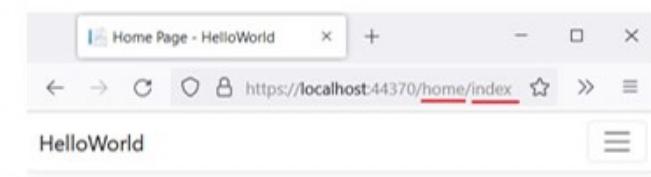
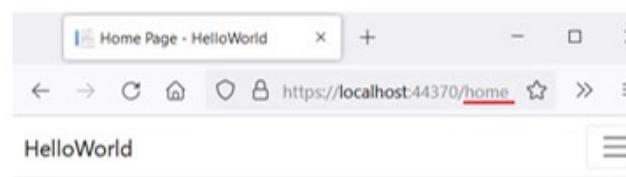
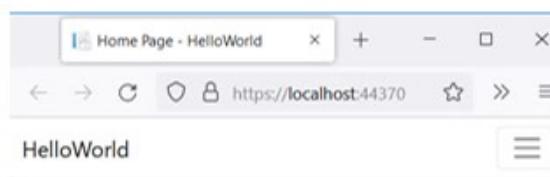
```
public IActionResult Hello(int id)  
{  
    if (id > 10)  
        return Json(new { description = "bla bla bla" });  
    else if (id > 5)  
        return NotFound();  
    else  
        return Content("alguma coisa correu mal...");  
}
```



## WEB | Routes (1/2)

- As rotas são definidas no ficheiro no arranque do programa através do ficheiro `program.cs`. Quando da criação de uma nova aplicação, é automaticamente adicionada uma rota *default*.
- Analise o excerto de código apresentado na imagem. Repare que a rota *default* espera receber o nome do controller, seguido do nome da action, e um id que é opcional. Note também que são definidos os valores por omissão, que serão utilizados caso não seja dado o valor para o controller / action (i.e., Home e Index respetivamente).
- Considerando os valores por omissão definidos na rota, analise as imagens abaixo para entender que, qualquer um dos exemplos, aponta sempre para a mesma View.

```
Program.cs
16 app.UseHttpsRedirection();
17 app.UseStaticFiles();
18
19 app.UseRouting();
20
21 app.UseAuthorization();
22
23 app.MapControllerRoute(
24     name: "default",
25     pattern: "{controller=Home}/{action=Index}/{id?}");
26
27 app.Run();
```



Learn about [building Web apps with ASP.NET Core.](#)

Learn about [building Web apps with ASP.NET Core.](#)

Learn about [building Web apps with ASP.NET Core.](#)

continua no slide seguinte...

## WEB | Routes (2/2)

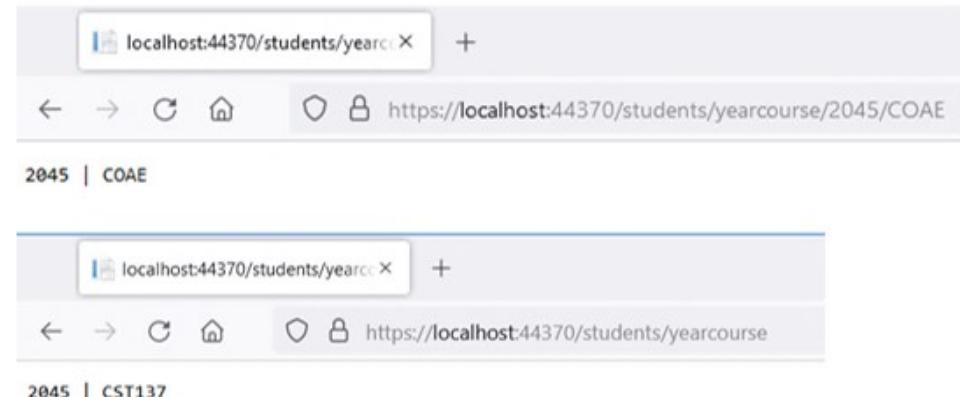
- Pode ser necessário adicionar outra rota, que não obedeça à estrutura da rota definida como *default*. Por exemplo, pode ser necessário receber dois ou mais argumentos de entrada à URL do *request*. No exemplo apresentado abaixo, define-se uma rota personalizada que recebe dois argumentos de entrada na URL.

```
app.MapControllerRoute(
    name: "StudentsYearCourse",
    pattern: "Students/YearCourse/{year=2045}/{course=CST137}",
    new { controller = "Students", action = "YearCourse" }
);

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

public ActionResult YearCourse(int year, string course)
{
    return Content($"{ year.ToString() } | { course }");
}
```



- No exemplo abaixo adiciona-se uma expressão regular para impor o cumprimento de regras num determinado valor de parâmetro de entrada.

```
app.MapControllerRoute(
    name: "StudentsYearCourse",
    pattern: "Students/YearCourse/{year=2045}/{course=CST137}",
    new { controller = "Students", action = "YearCourse" },
    new { year = @"^([0-9]{4})$" }
);
```

## WEB | Debug do Visual Studio

- O *debug* também está disponível no desenvolvimento de aplicações em Razor. Implemente o código seguinte, adicione um *breakpoint* numa linha de código, e experimente executar passo a passo.

```
public class WelcomeController : Controller
{
    0 references
    public IActionResult Hello(int id)
    {
        if (id > 10)
            return Json(new { description = "bla bla bla" });
        else if (id > 5)
            return NotFound();
        else if (id == 1)
        {
            ViewBag.Number = 1234;
            return View();
        }
        else
            return Content("alguma coisa correu mal...");
    }
}
```



## ViewBag example

The number that came from the controller was 1234

## WEB | Ficheiro \_Layout.cshtml

- O ficheiro `_Layout.cshtml` funciona como uma *masterpage* da aplicação. Funciona como um modelo de layout, definindo a estrutura comum para todas as páginas da aplicação (pode haver mais que um destes ficheiros).
- Funciona como um esqueleto para outras páginas, dando-lhes a estrutura básica (e.g. menu, rodapé). Através desta estratégia, além de se conseguir manter a consistência visual ao longo das várias funcionalidades da aplicação, também se promove a não repetição de código / *markup*. Também facilita a manutenção do código, já que uma alteração no `_Layout.cshtml` tem impacto em toda a aplicação.
- O `_Layout.cshtml` pode conter código Razor para incluir conteúdos dinâmicos na estrutura (se necessário). As Views da aplicação, serão apresentadas em vez da instrução `@RenderBody()`.

```
<body>
  <header>
    <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
      <div class="container-fluid">
        <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">HelloWorld</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
          <ul class="navbar-nav flex-grow-1">
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
            </li>
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
            </li>
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-action="Get">Students</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </header>
  <div class="container">
    <main role="main" class="pb-3">
      @RenderBody()
    </main>
  </div>
  <footer class="border-top footer text-muted">
    <div class="container">
      ...
    </div>
  </footer>

```

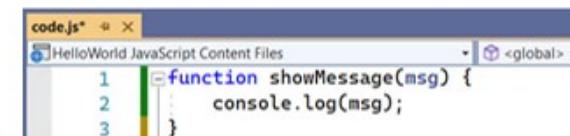
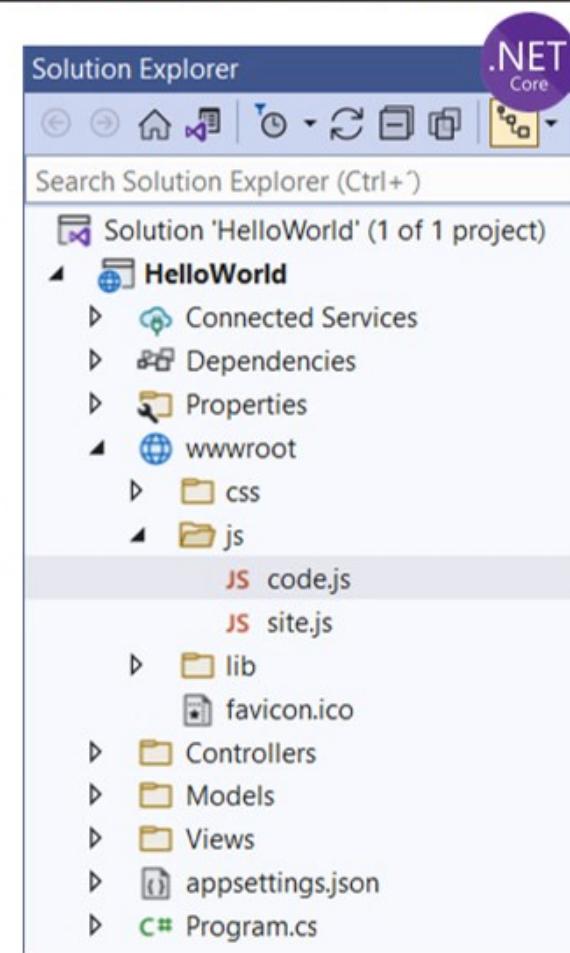
## WEB | Inclusão de JavaScript

- Nos projetos ASP.Net Web é possível adicionar ficheiros com código JavaScript. Os ficheiros devem ser guardados dentro da pasta `wwwroot` e, posteriormente, referenciados no HTML para serem transferidos para o lado do cliente.
- No exemplo apresentado nas imagens é criado o ficheiro `code.js` dentro da pasta `wwwroot\js`. Posteriormente, e para que o ficheiro fique disponível globalmente na aplicação, o ficheiro é referenciado no HTML através do ficheiro `_Layout.cshtml`. Inclui-se também um elemento HTML para executar um função existente no ficheiro `code.js`.



```
43 </footer>
44 <script src="~/lib/jquery/dist/jquery.min.js"></script>
45 <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
46 <script src="~/js/site.js" asp-append-version="true"></script>
47 <script src="~/js/code.js" asp-append-version="true"></script>
48 @await RenderSectionAsync("Scripts", required: false)
49 </body>
50 </html>

<div>
    <a href="#" onclick="showMessage('this is a simple test message!');">click me</a>
</div>
```



```
function showMessage(msg) {
    console.log(msg);
}
```

## WEB | Partial Views

- As *Partial Views* funcionam como pequenas partes reutilizáveis noutras *Views*. As *Partial Views* funcionam como *WebUserControls* da tecnologia de desenvolvimento *web* anterior do .NET. É boa prática iniciar o nome das *Partial Views* com *underscore* (\_), significando tratar-se de um ficheiro parcial a ser utilizado (referenciado) noutras *Views*.
- A imagem abaixo apresenta um bloco HTML que foi movido para dentro de uma *Partial View*. Posteriormente, pode ser referenciado apenas com a tag especial chamada `<partial />`.

```

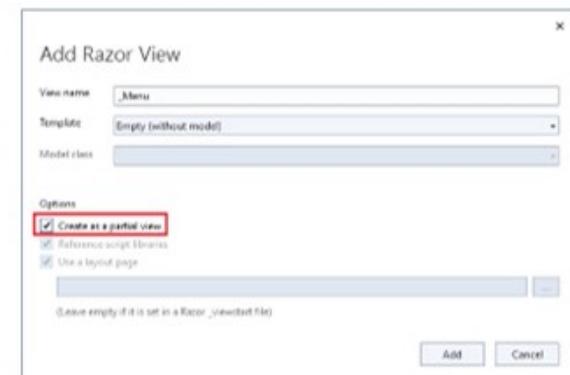
<body>
  <header>
    <nav class="navbar navbar-expand-sm navbar-light bg-white border-bottom box-shadow mb-3">
      <div class="container">
        <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">HelloWorld</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <partial name="_Menu"/>
      </div>
    </nav>
  </header>
<div class="content">

```

```

<body>
  <header>
    <nav class="navbar navbar-expand-sm navbar-light bg-white border-bottom box-shadow mb-3">
      <div class="container">
        <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">HelloWorld</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
          <ul class="navbar-nav flex-grow-1">
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
            </li>
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
            </li>
          </ul>
          <partial name="_LoginPartial" />
        </div>
      </div>
    </nav>
  </header>
<div class="content">

```



## WEB | Tag Helpers

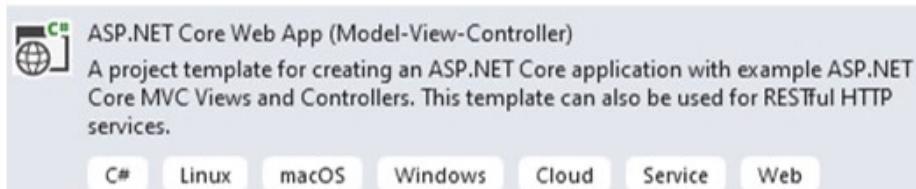
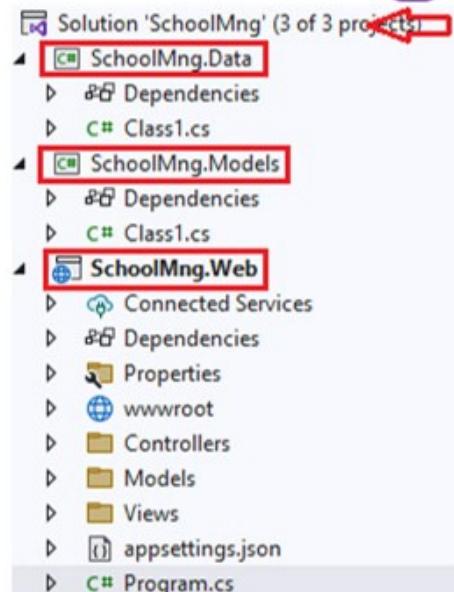
- Em ASP.NET Core podem utilizar-se e criar-se *tags* HTML personalizadas. Estas *tags* processam código do lado do servidor, e geram HTML automaticamente.
- Os *Tag Helpers* permite simplificar o desenvolvimento da solução, embebendo no HTML tradicional, indicações a serem processadas pelo servidor. Pretende-se tornar o código / *markup* mais fácil de ler.
- Os *Tag Helpers* são usados para estender as funcionalidades das *tags* HTML tradicionais, e facilitam a integração / *binding* dos dados.
- Na imagem abaixo, a título de exemplo, salientou-se a vermelho alguns exemplos de *tag helpers*. No exemplo, a *tag* `asp-controller` indica que o `form` deve ser submetido ao controller `Students`, onde deve ser executado a *action* `Create`.

```
@model Teacher

<form class="form-horizontal" method="post" asp-controller="Students" asp-action="Create">
    <label class="form-label" asp-for="Name"></label>
    <input class="form-control" type="text" />
</form>
```

## WEB | Demonstração SchoolMng (1/17)

- Nesta demonstração vai ser criada uma pequena aplicação para gerir os Teacher e Student de uma escola. Vai ser utilizado o *EntityFramework* (EF) como camada de acesso a dados em SQL Server (pode utilizar outro tipo de base de dados se preferir).
- Crie uma solução chamada SchoolMng, e inicie com um projeto chamado SchoolMng.Web (template ASP.NET Web MVC). Inclua também um projeto chamado SchoolMng.Models onde serão especificados os *Models* do projeto, e um projeto SchoolMng.Data que será o projeto para acesso a dados. Estes dois últimos projetos serão do tipo Class Library.
- Note que o nome dado aos vários projetos são apenas sugestões. Poderia optar-se por outra estratégia de *naming*.



continua no slide seguinte...



## WEB | Demonstração SchoolMng (2/17)

- Adicione os *Nuget packages* do quadro abaixo aos projetos indicados.

Nuget package	Projetos
Microsoft.EntityFrameworkCore.Design	SchoolMng.Data; SchoolMng.Web
Microsoft.EntityFrameworkCore.SqlServer	SchoolMng.Data; SchoolMng.Web
Microsoft.EntityFrameworkCore.Tools	SchoolMng.Data; SchoolMng.Web

- Adicione a *connection string* SchoolMngConnection. Siga a imagem abaixo, mas não esqueça de adequar os dados ao seu ambiente de desenvolvimento (e.g instância SQL Server).

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Warning"  
    }  
  },  
  "AllowedHosts": "*",  
  "ConnectionStrings": {  
    "SchoolMngConnection": "server=.;database=SchoolMng;Trusted_Connection=True;TrustServerCertificate=True;"  
  }  
}
```

continua no slide seguinte...

## WEB | Demonstração SchoolMng (3/17)

- Adicione ao projeto SchoolMng.Models as classes apresentadas nas imagens. Note a utilização de *Annotations* que especificam regras sobre as propriedades dos *Models*.
- As regras definidas nos *Models*, serão utilizadas pelo EF para a definição do *schema* da base de dados (BD), mas também serão utilizadas pelo ASP.NET Core na apresentação e validação de dados dos *interfaces*.
- Notar que, por questões de economia de espaço deste documento, não se utilizará a comentação de código. No entanto, num ambiente real, a comentação deverá existir.

```
public class Teacher
{
    public int Id { get; set; }

    [Required]
    [StringLength(50)]
    [RegularExpression(@"^([A-Z]{2,50}$)")]
    [Display(Name="Name (fullname)")]
    public string? Name { get; set; }
}

public class Student
{
    [Key]
    public int Number { get; set; }

    [Required, StringLength(50)]
    public string Name { get; set; }

    [Required]
    [DataType(DataType.ImageUrl)]
    public string? Photo { get; set; }

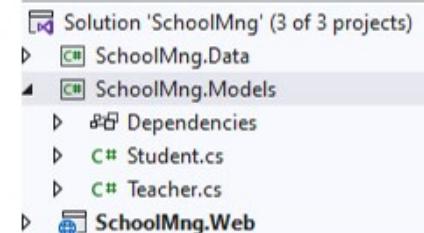
    [Required, Range(18, 100)]
    public int Age { get; set; }

    [Required, DataType(DataType.Currency)]
    public float Fee { get; set; }

    [Required, DataType(DataType.DateTime), DisplayFormat(DataFormatString = "{0: yyyy-MM-dd}")]
    public DateTime CreationDate { get; set; }

    [Required, ForeignKey("Teacher")]
    public int TeacherId { get; set; }

    public Teacher? Teacher { get; set; }
}
```



continua no slide seguinte...

## WEB | Demonstração SchoolMng (4/17)

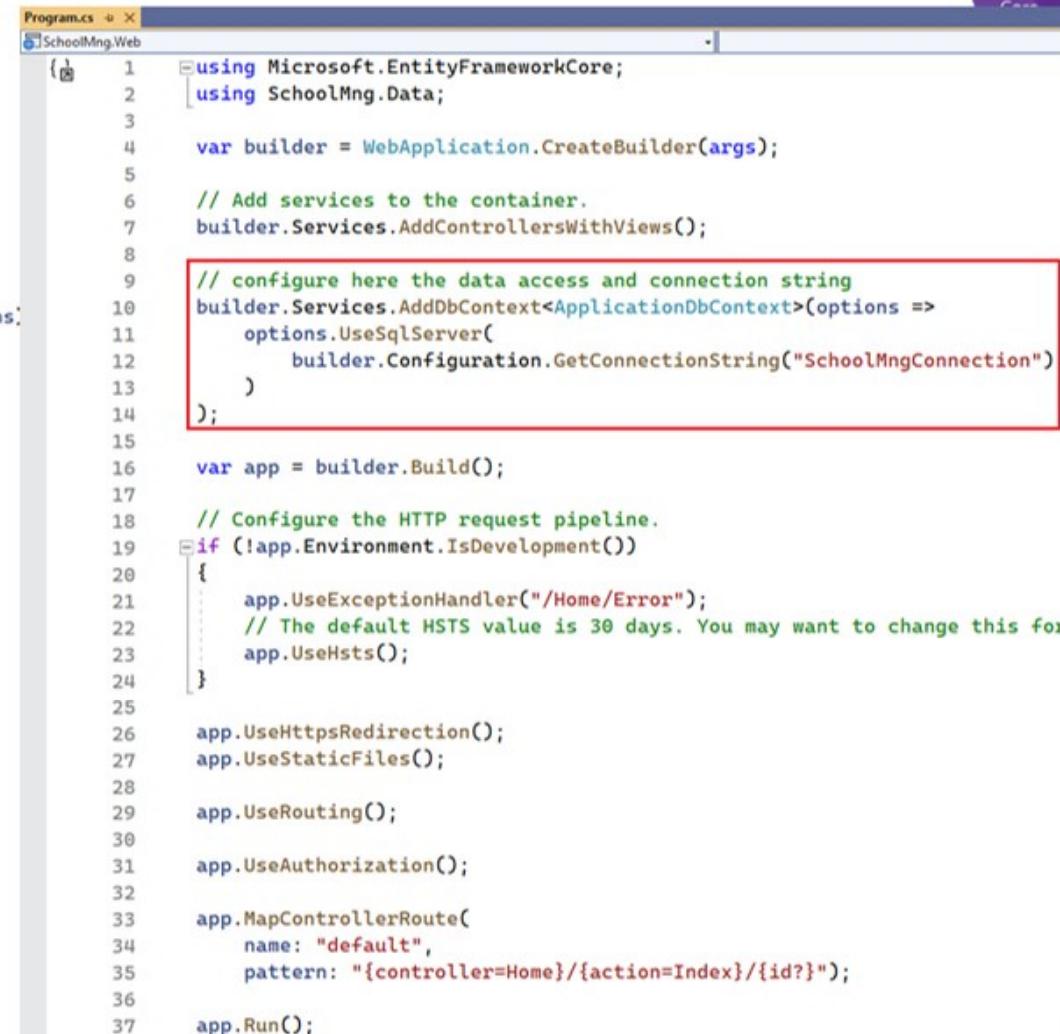
- Implemente o DbContext do EF no projeto SchoolMng.Data. Considerando a imagem abaixo, crie a classe ApplicationContext.

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

    0 references
    public DbSet<Teacher> Teachers { get; set; }

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

- Depois de implementar a classe ApplicationContext, ative o serviço EF no ficheiro program.cs. Atenda ao local correto onde implementar o código para incluir o serviço de código (ver imagem ao lado).
- Crie o *schema* da BD utilizando *migrations* do EF.

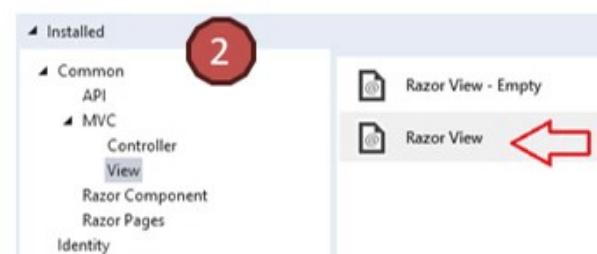
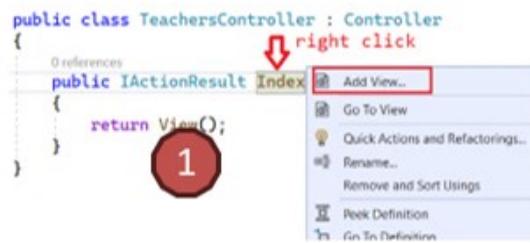


```
Program.cs  ✘ SchoolMng.Web
1  using Microsoft.EntityFrameworkCore;
2  using SchoolMng.Data;
3
4  var builder = WebApplication.CreateBuilder(args);
5
6  // Add services to the container.
7  builder.Services.AddControllersWithViews();
8
9  // configure here the data access and connection string
10 builder.Services.AddDbContext<ApplicationContext>(options =>
11     options.UseSqlServer(
12         builder.Configuration.GetConnectionString("SchoolMngConnection")
13     )
14 );
15
16 var app = builder.Build();
17
18 // Configure the HTTP request pipeline.
19 if (!app.Environment.IsDevelopment())
20 {
21     app.UseExceptionHandler("/Home/Error");
22     // The default HSTS value is 30 days. You may want to change this for
23     app.UseHsts();
24 }
25
26 app.UseHttpsRedirection();
27 app.UseStaticFiles();
28
29 app.UseRouting();
30
31 app.UseAuthorization();
32
33 app.MapControllerRoute(
34     name: "default",
35     pattern: "{controller=Home}/{action=Index}/{id?}");
36
37 app.Run();
```

continua no slide seguinte...

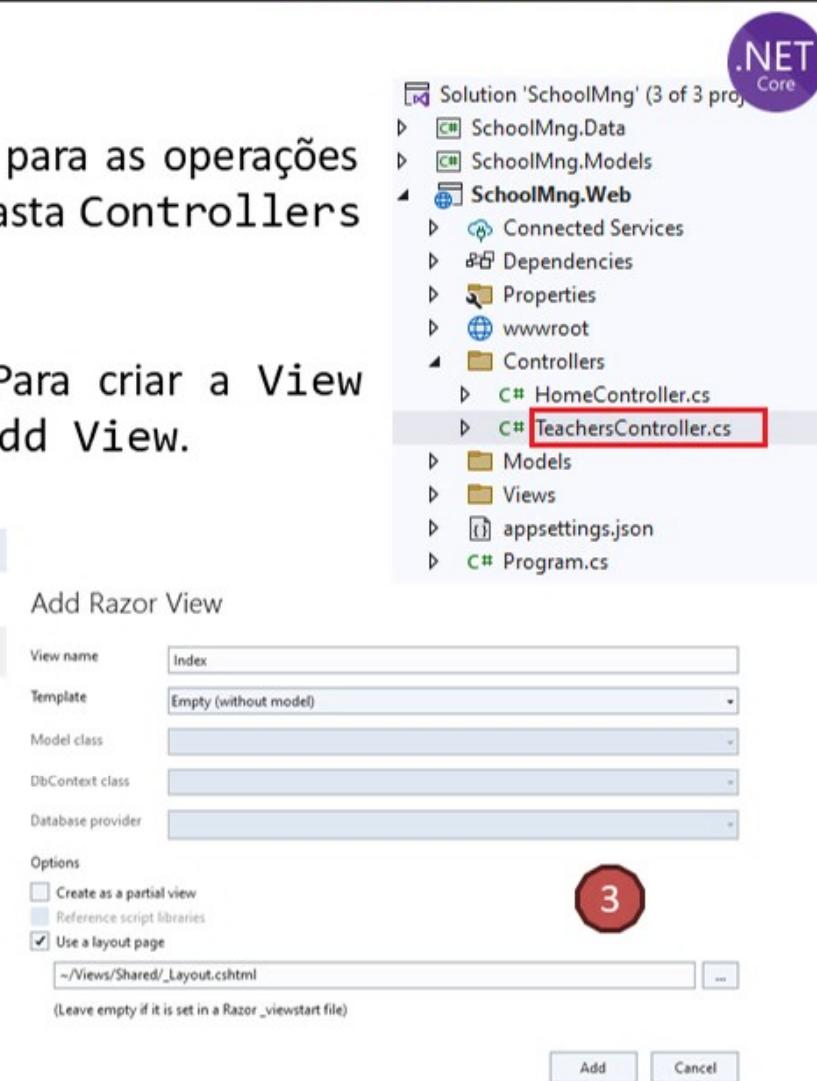
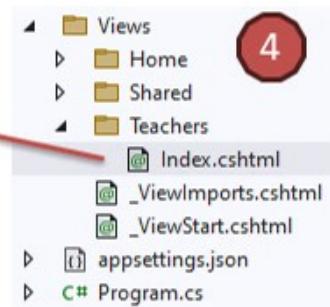
## WEB | Demonstração SchoolMng (5/17)

- De seguida, vai criar-se o Controller que ficará com as *actions* para as operações CRUD dos Teachers. Adicione um novo Controller dentro da pasta Controllers seguindo as imagens abaixo.
- Repare que foi adicionada a *action* Index automaticamente. Para criar a View correspondente, utilize o botão direito do rato e selecione a opção Add View.



```
@{
    ViewData["Title"] = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h1>Index</h1>
```

A red arrow labeled '5' points from the 'Index' view code back to the 'Index' action method in the controller code.



continua no slide seguinte...

## WEB | Demonstração SchoolMng (6/17)

- De seguida, esta nova opção será adiciona ao menu da aplicação. Edite o ficheiro `_Layout.cshtml` e adicione o conteúdo assinalado abaixo.

```
</button>
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
  <ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Teachers" asp-action="Index">Teachers</a>
    </li>
  </ul>
```

- Voltando ao *endpoint* `Index` criado dentro do Controller Teacher. Pretende-se que a View apresente os Teacher existentes na base de dados. Para isso, vai utilizar-se EF para se conseguir obter estes dados. Implemente o código abaixo para ter disponível dentro do controller o acesso ao `DbContext` do EF.

```
public class TeachersController : Controller
{
  private readonly ApplicationDbContext context;
  0 references
  public TeachersController(ApplicationDbContext dbContext)
  {
    this.context = dbContext;
  }
}
```

```
0 references
public IActionResult Index()
{
  return View();
}
```

continua no slide seguinte...

## WEB | Demonstração SchoolMng (7/17)

- Após ligar o Controller ao EF, carregue-se a lista de Teacher da base de dados para ser enviada à View. Altere a *action* em conformidade com a imagem.
- A View recebe esta *collection* no argumento Model (atenção que a primeira letra é maiúscula). A receção do argumento é definido com @model (atenção que a primeira letra é minúscula). Implemente o código do Index.cshtml da imagem.
- Note que, para que a classe Teacher seja encontrada dentro da View, tem que se utilizar a diretiva using para importar a *namespace*, neste caso, foi incluída no ficheiro \_ViewImports.cshtml para que o *namespace* SchoolMng.Models fique visível em todas as Views.
- Execute a aplicação e valide se os registos são apresentados. Insira dados diretamente na tabela para obter resultados.

continua no slide seguinte...

The screenshot shows a .NET Core application interface. At the top right is a purple ".NET Core" logo. Below it are two code snippets. The first snippet is a C# controller action named 'Index':

```
public IActionResult Index()
{
    List<Teacher> teachers = context.Teachers.ToList();
    return View(teachers);
}
```

Two red arrows point from the text "conforme a imagem." in the slide to the opening brace of the method and the closing brace of the return statement. The second snippet is an ASP.NET Core view file named 'Index.cshtml':

```
@* @using SchoolMng.Models *@
@model List<Teacher>

@{
    ViewData["Title"] = "Teachers list";
    Layout = "~/Views/Shared/_Layout.cshtml";
}



# Teachers



| Number | Name |
|--------|------|
|--------|------|


```

The third part of the screenshot shows the '\_ViewImports.cshtml' file in the file browser:

```
1 @using SchoolMng.Web
2 @using SchoolMng.Web.Models
3 @using SchoolMng.Models
4 @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

A red arrow points from the text "conforme a imagem." in the slide to the line '@addTagHelper \*,' Microsoft.AspNetCore.Mvc.TagHelpers'.

## WEB | Demonstração SchoolMng (8/17)

- De seguida implemente-se a opção de criar um novo Teacher na tabela. De volta ao controller Teachers, implemente a *action* Create() como apresentado na imagem. Note que, nesta *action*, não está a ser passado nenhum argumento de entrada para a View.
- Utilize uma vez mais o botão direito do rato, ou crie manualmente a View Create.cshtml dentro da pasta View\Teachers. Repare que, apesar de não estar a ser enviado nenhum valor, está a ser declarado um argumento do tipo Teacher. Por defeito, se nenhum valor for enviado, é criado um objeto novo (neste caso é o que se precisa).
- Antes ainda de se implementar o resto do conteúdo do ficheiro Create.cshtml, vai-se adicionar um botão para chamar a View Create, para quando se quiser criar um Teacher novo. Abra a View Index.cshtml e incorpore o conteúdo assinalado a vermelho.
- Execute a aplicação para testar a navegabilidade para o novo formulário.

continua no slide seguinte...

```
// HTTP Method: GET
0 references
public IActionResult Create()
{
    return View();
}

@model Teacher ←
@{
    ViewData["Title"] = "Create";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>Create</h1>

}
<h1>Teachers</h1>

<div>
    <a asp-controller="Teachers" asp-action="Create" class="btn btn-primary">
        Create New Teacher
    </a>
</div>

<table class="table table-bordered table-striped">
    <tr>
```

## WEB | Demonstração SchoolMng (9/17)

- A imagem ao lado apresenta a implementação completa da View Create.cshtml.
- Notar a utilização dos *Tag Helpers* para mapear as propriedades do objeto Teacher para as tags do HTML (*binding*). Notar também que o método de submissão do form é o POST.
- De seguida, falta criar a *action* no controller Teachers que será executado quando for feito o POST do form. Implemente o código da imagem.
- A *annotation* [ValidateAntiForgeryToken] é um mecanismo automático que valida que a *action* esteja a ser executada a partir da View Create apresentada ao utilizador.
- O método recebe o objeto Teacher enviado pela View, e guarda-o na base de dados utilizando o EF. Teste o funcionamento da aplicação após a implementação.

```
@model Teacher
{
    ViewData["Title"] = "Create";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>Create</h1>

<form method="post">
    <div class="border m-2 p-4">
        <div class="row pb-2">
            <h2 class="text-primary">Create Teacher</h2>
        </div>
        <div class="mb-3">
            <label asp-for="Name" class="form-label"></label>
            <input asp-for="Name" class="form-control" />
        </div>
        <button type="submit" class="btn btn-primary">Create</button>
        <a asp-controller="Teachers" asp-action="Index" class="btn btn-secondary">Back</a>
    </div>
</form>
```

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public IActionResult Create(Teacher teacher)
{
    // perform validations here...
    // (...) stay tuned...

    context.Teachers.Add(teacher);
    context.SaveChanges();
    //return RedirectToAction("Index", "Teachers");
    return RedirectToAction("Index");
}
```

continua no slide seguinte...

## WEB | Demonstração SchoolMng (10/17)

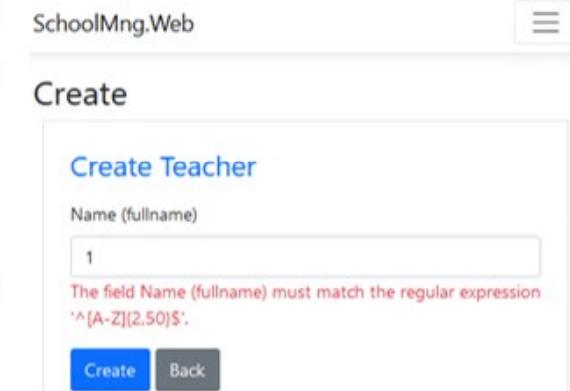
- Falta apenas adicionar as validações do objeto recebido. Antes de enviar os dados para a BD através do EF, tem que se validar o conteúdo das propriedades. Implemente o código da imagem, para garantir que o conteúdo do objeto é congruente com as regras definidas no *Model*.
- A instrução adicionada, vai garantir que a gravação só é efetuada se o *Model* for válido. Para ser possível ver o erro no formulário, adicione o conteúdo na View Create.cshtml.
- De seguida, execute a aplicação e tente guardar um Teacher com um nome inválido, considerando as regras definidas no Model (e.g. tente guardar um Teacher com uma letra apenas no nome).
- Por fim, personalize o erro que deve surgir ao utilizador, alterando as *annotations* do *Model* para a imagem apresentada abaixo. Depois teste novamente.

```
[Required]
[StringLength(50)]
[RegularExpression(@"^([A-Z]{2,50}$",
    ErrorMessage = "Content does not respect the rules...")]
[Display(Name="Name (fullname)")]
4 references
public string? Name { get; set; }
```

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public IActionResult Create(Teacher teacher)
{
    // validations
    if (!ModelState.IsValid)
        return View(teacher);

    context.Teachers.Add(teacher);
}

<div class="mb-3">
    <label asp-for="Name" class="form-label"></label>
    <input asp-for="Name" class="form-control" />
    <span asp-validation-for="Name" class="text-danger"></span>
</div>
<button type="submit" class="btn btn-primary">Create</button>
```



continua no slide seguinte...

## WEB | Demonstração SchoolMng (11/17)

- Os erros podem ser apresentados campo a campo, e/ou num summary. Altere o conteúdo do Create.cshtml em conformidade com a imagem.
- Além das validações efetuadas tendo por base as regras definidas nos Models, é também possível adicionar verificações. Altere o código da action Create de acordo com a imagem.
- Repare que é possível prever validações adicionais, e adicioná-las ao pipeline de processamento / apresentação de erros do ASP.Net Core. Se os erros adicionados forem relacionados com uma propriedade, a mensagem será apresentada próximo do campo correspondente na interface.

continua no slide seguinte...

```

<div class="row pb-2">
    <h2 class="text-primary">Create Teacher</h2>
</div>
<div asp-validation-summary="All"></div>
<div class="mb-3">
    <label asp-for="Name" class="form-label"></label>
    <input asp-for="Name" class="form-control" />
</div>

// HTTP Method: POST (handle the submit button of the form)
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Teacher teacher)
{
    // name of the teacher can not be "Manel"
    if (teacher.Name.Equals("Manel"))
        ModelState.AddModelError("Error 001", "The name 'Manel' can only be used by Célio!!");

    // max number of Teachers: 4
    if (_db.Teachers.Count() >= 4)
        ModelState.AddModelError("Name", "Max number of Teachers was reached!");

    // validate the model rules
    if (!ModelState.IsValid)
        return View(teacher);

    // if everything is ok, save the data into db
    _db.Teachers.Add(teacher);
    _db.SaveChanges(); // save in the base
    //return RedirectToAction("Index", "Teachers");
    return RedirectToAction("Index");
}

```

- The content does not respect the rules...
- The name 'Manel' can only be used by Célio!!
- Max number of Teachers was reached!

The content does not respect the rules...

Max number of Teachers was reached!

## WEB | Demonstração SchoolMng (12/17)

- O ASP.Net Core também permite configurar, de uma forma simples, que a validação das regras do *Model* passem para o lado do cliente (JavaScript). Altere o `Create.cshtml` acrescentando o conteúdo salientado na imagem e execute novamente a aplicação.
- Depois de executar a aplicação, inspecione no *browser* os elementos do conteúdo recebido do servidor. Note a adição de conteúdo para tornar possível a validação do lado do cliente. Simule um erro no preenchimento do nome, e valide que a verificação do erro é feito do lado do cliente (*browser*).

continua no slide seguinte...



```
1  @model Teacher
2  @{
3      ViewData["Title"] = "Create";
4      Layout = "~/Views/Shared/_Layout.cshtml";
5  }
6
7  <h1>Create</h1>
8
9  <form method="post">
10     <div class="border m-2 p-4">
11         <div class="row pb-2">
12             <h2 class="text-primary">Create Teacher</h2>
13         </div>
14         <div asp-validation-summary="All"></div>
15         <div class="mb-3">
16             <label asp-for="Name" class="form-label"></label>
17             <input asp-for="Name" class="form-control" />
18             <span asp-validation-for="Name" class="text-danger"></span>
19         </div>
20         <button type="submit" class="btn btn-primary">Create</button>
21         <a asp-controller="Teachers" asp-action="Index" class="btn btn-secondary">Back</a>
22     </div>
23 </form>
24
25 @section Scripts {
26     @{
27         <partial name="_ValidationScriptsPartial" />
28     }
29 }
```



Create

Create Teacher

Name (fullname)

a

Content does not respect the rules...

Create Back



## WEB | Demonstração SchoolMng (13/17)

- De seguida vai implementar-se a edição de objetos. Neste caso, a *action* Edit deverá receber um *id* aquando da sua chamada. Este argumento não pode ser null, e vai ser usado para carregar os dados do Teacher correspondente. Implemente a *action* Edit(int) como apresentado na imagem.
- Analise as várias formas de encontrar / carregar o registo a editar utilizando o EF. Consulte a documentação do EF, para analisar as diferenças existentes entre os métodos FirstOrDefault(), SingleOrDefault() e Find().
- Analise também os vários *HttpCodes* que são devolvidos em cada situação. Utilizam-se *Helper Methods* para devolver informação útil ao cliente. O BadRequest() devolve um *HttpCode* 400, e o NotFound() devolve um *HttpCode* 404. Para mais informações consulte o link abaixo.
  - <https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-7.0>

```
// HTTP Method: GET
0 references
public IActionResult Edit(int? id)
{
    Teacher teacher;

    if (id == null)
        return BadRequest();

    if (id <= 0)
        return NotFound();

    //teacher = context.Teachers.FirstOrDefault(t => t.Id == id);
    //teacher = context.Teachers.SingleOrDefault(t => t.Id == id);
    teacher = context.Teachers.Find(id);

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

    return View(teacher);
}
```

continua no slide seguinte...



## WEB | Demonstração SchoolMng (14/17)

- Implemente a View correspondente ao Edit, utilizando o botão direito do rato como fez na View Create. Implemente o conteúdo do Edit.cshtml como apresentado na imagem.
- Neste exemplo, está a utilizar-se *Tag Helpers* na tag do form. Neste caso não seriam necessárias porque, quando não são especificadas, são assumidos o controller e action que carregou a View. Resolveu-se por utilizá-las nesta solução, apenas para salientar que existem e que podem ser utilizadas se necessário.
- Note que o valor da propriedade Id, neste formulário, é apresentado. No entanto, a tag do input foi decorado com o atributo disabled, porque não é suposto ser alterado pelo utilizador (chave primária).

```
@model Teacher

 @{
    ViewData["Title"] = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>Edit Teacher</h1>

<form method="post" asp-controller="Teachers" asp-action="Edit">
    <div class="border m-2 p-4">
        <div class="row pb-2">
            <h2 class="text-primary">Edit Teacher</h2>
        </div>
        <div asp-validation-summary="All"></div>
        <div class="mb-3">
            <label asp-for="Id" class="form-label"></label>
            <input asp-for="Id" class="form-control" disabled />
        </div>
        <div class="mb-3">
            <label asp-for="Name" class="form-label"></label>
            <input asp-for="Name" class="form-control" />
            <span asp-validation-for="Name" class="text-danger"></span>
        </div>
        <button type="submit" class="btn btn-primary">Update</button>
        <a asp-controller="Teachers" asp-action="Index" class="btn btn-secondary">Back</a>
    </div>
</form>

@section Scripts {
    @{
        <partial name="_ValidationScriptsPartial" />
    }
}
```

continua no slide seguinte...

## WEB | Demonstração SchoolMng (15/17)

- De seguida, falta criar a *action* no controller Teachers que será executado quando for feito o POST do form, com a nova versão do objeto. O código da imagem apresenta a *action* Edit(Teacher) responsável por atualizar a BD.
- De seguida, é necessário testar o funcionamento do Edit. No entanto, é ainda necessário colocar o botão de edição na lista de Teachers, para que seja possível fazer a edição.
- Edite o ficheiro Index.cshtml dos Teachers e adicione o *markup* indicado na imagem, responsável por apresentar o botão Edit na última coluna da lista apresentada.
- Execute a aplicação e garanta que os dados são atualizados na base de dados.

continua no slide seguinte...

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public IActionResult Edit(Teacher teacher)
{
    // causing some extra errors
    if (teacher.Name.Equals("MANEL"))
        ModelState.AddModelError("Error 0001", "Not allowed...");

    // validations
    if (!ModelState.IsValid)
        return View(teacher);

    // update data
    context.Teachers.Update(teacher);
    context.SaveChanges();
    return RedirectToAction("Index");
}

<tr>
    <th>Number</th>
    <th>Name</th>
    <th>...</th>
</tr>

@foreach (Teacher teacher in Model)
{
    <tr>
        <td>@teacher.Id</td>
        <td>@teacher.Name</td>
        <td>
            <a asp-controller="Teachers" asp-action="Edit" asp-route-id ="@teacher.Id">Edit</a>
        </td>
    </tr>
}
```

## WEB | Demonstração SchoolMng (16/17)

- Por último, vai-se agora implementar a funcionalidade da remoção de objetos da BD. O código da *action* Delete() é similar ao da *action* implementada anteriormente. Implemente o método apresentado na imagem no controller Teachers, que carrega o Teacher a remover, e apresenta uma View com a confirmação da remoção.

- Implemente a View correspondente ao Delete. Implemente o conteúdo do Delete.cshtml como apresentado na imagem.

- Note que, neste caso, os *input* estão todos disabled. Note também que foi incluído um *input hidden* com o Id do objeto em edição. A *action* que vai ser executada no controller quando este form for submetido, receberá apenas um inteiro que será este int.

continua no slide seguinte...



```
// HTTP Method: GET
0 references
public IActionResult Delete(int? id)
{
    Teacher teacher;

    if (id == null)
        return BadRequest();

    if (id <= 0)
        return NotFound();

    teacher = context.Teachers.Find(id);

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

    return View(teacher);
}
```

```
@model Teacher

@{
    ViewData["Title"] = "Delete";
    Layout = "~/Views/Shared/_Layout.cshtml";
}



# Delete Teacher



<form method="post" asp-action="DeletePost">
    <input asp-for="Id" hidden />
    <div class="border m-2 p-4">
        <div class="row pb-2">
            <h2 class="text-primary">Delete Teacher</h2>
        </div>
        <div asp-validation-summary="All"></div>
        <div class="mb-3">
            <label asp-for="Id" class="form-label"></label>
            <input asp-for="Id" class="form-control" disabled />
        </div>
        <div class="mb-3">
            <label asp-for="Name" class="form-label"></label>
            <input asp-for="Name" class="form-control" disabled />
        </div>
        <button type="submit" class="btn btn-primary">Delete</button>
        <a asp-controller="Teachers" asp-action="Index" class="btn btn-secondary">Back</a>
    </div>
</form>
```

## WEB | Demonstração SchoolMng (17/17)

- Como já indicado no slide anterior, a *action* que será executada no controller quando o form da View Delete for executada, recebe um inteiro. Como o Delete() que é chamado para carregar a View tem a mesma assinatura, não é possível fazer *overloading* dos métodos por terem a mesma assinatura.
- Por este motivo, a *action* que será consumida com o POST foi renomeada para DeletePost(). Foi, também, incluída a *tag helper* asp-action no elemento form, para indicar qual a *action* do controller que deve ser executada aquando do POST do form.
- Por fim, adicione o botão remove no Index.cshtml para que esteja disponível para cada Teacher e teste a aplicação.

```
<tr>
    <td>@teacher.Id</td>
    <td>@teacher.Name</td>
    <td>
        <a asp-controller="Teachers" asp-action="Edit" asp-route-id ="@teacher.Id">Edit</a>
        <a asp-controller="Teachers" asp-action="Delete" asp-route-id="@teacher.Id">Delete</a>
    </td>
</tr>
```

```
[HttpPost, ActionName("DeletePost")]
[ValidateAntiForgeryToken]
0 references
public IActionResult DeleteTeacher(int? id)
{
    Teacher teacher = context.Teachers.Find(id);

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

    context.Teachers.Remove(teacher);
    context.SaveChanges();
    return RedirectToAction("Index");
}
```



## WEB | ViewBag, ViewData, TempData

- O ViewBag, ViewData, e TempData, são mecanismos disponibilizados pelo ASP.Net Core para transmitir dados entre Controllers e Views, Controllerse Controllers, e Actions para Actions.
- ViewBag
  - Permite a transmissão de dados através da criação de propriedades *on-the-fly*.
  - Tipo de dados `dynamic` (atribuído aquando da atribuição do valor).
  - Vive apenas durante o *request* que é criado (morre com o *response*).
  - Exemplo de utilização: `ViewBag.Nome = "Célio Carvalho";`

```
public class TestController : Controller
{
    public IActionResult Index()
    {
        ViewBag.NumberOfUsers = 56;
        return View();
    }
}
```

```
<h1>
    ViewBag test
</h1>

<p>ViewBag NumberOfUsers is @ViewBag.NumberOfUsers</p>
<p>ViewBag is discarded after this response.</p>
```

## WEB | ViewBag, ViewData, TempData

- **ViewData**

- Bastante utilizado na passagem de dados entre o Controller para a View.
- A diferença principal para o ViewBag é que este objeto é tipo Dictionary<key, value>.
- O tipo de dados é Dictionary<string, object?>.
- Vive apenas durante o *request* que é criado (morre com o *response*).
- Exemplo de utilização: ViewData[“nome”] = “Célio Carvalho”.

```
public class TestController : Controller
{
    0 references
    public IActionResult Index()
    {
        ViewData["NumberOfUsers"] = 56;
        return View();
    }
}

<h1>
    ViewData test
</h1>

<p>ViewData NumberOfUsers is
    @{
        @(int.Parse(ViewData["NumberOfUsers"].ToString()) + 10000)
    }
</p>
<p>ViewData is discarded after this response.</p>
```



## WEB | ViewBag, ViewData, TempData

- TempData (1/2)
  - Diferente dos anteriores, porque permite passar dados entre 2 (ou mais) *requests*. Os dados são guardados pelo ASP.Net dentro da *session* da sua session.
  - O tipo de dados é `Dictionary<string, object?>`.
  - Vive apenas até ao próximo *request* (e não apenas no *request* atual como no `ViewBag` e `ViewData`).
  - Apesar de viver naturalmente até ao próximo *request*, é possível prolongar o seu tempo de vida utilizando os métodos `Keep()` e `Peek()`.
  - O método `Peek()` permite aceder ao valor do `TempData`. O `Keep()` também acede ao valor do `TempData`, mas indica ao ASP.Net que deve manter o valor por mais um *request*. Se utilizado o `Peek()` o valor será removido no próximo *request*.

### Utilização sem `Keep()` nem `Peek()`:

```
// request #1 (add object to dictionary)
TempData["name"] = "Manel";

// request #2 (read object from dictionary)
// the element exists, but is marked to be removed when readed
string name = TempData["name"]?.ToString() ?? "";

// request #3 (read object from dictionary)
// the element does not exists anymore in the dictionary
// TempData["name"] is null
```

### Utilização com `Peek()`:

```
// request #1 (add object to dictionary)
TempData["name"] = "Manel";

// request #2 (read object from dictionary)
// the element exists, and is marked to be removed when readed
string name = TempData.Peek("name")?.ToString() ?? "";

// request #3 (read object from dictionary)
// the element exists, but is marked to be removed when readed
string name = TempData["name"]?.ToString() ?? "";
```

### Utilização com `Keep()`:

```
// request #1 (add object to dictionary)
TempData["name"] = "Manel";

// request #2 (read object from dictionary)
// the element exists, but is marked to be removed when readed
string name = TempData["name"]?.ToString() ?? "";
// (...)

// later, in the same request, it can me marked to be kept
TempData.Keep("name");

// request #3 (read object from dictionary)
// the element exists, but is marked to be removed when readed
string name = TempData["name"]?.ToString() ?? "";
```

## WEB | ViewBag, ViewData, TempData

- TempData (2/2)

The diagram illustrates the state of TempData across three stages:

- Stage 1:** A red arrow points from the assignment of TempData["NumberOfUsers"] = 56; in the Action1() method to the TempData entry in the View code.
- Stage 2:** A red arrow points from the TempData entry in the View code to the TempData entry in the Action2() method's ViewData.
- Stage 3:** A red arrow points from the TempData entry in the Action2() method's ViewData back to the TempData entry in the View code.

**Action1() Method:**

```
public class TestController : Controller
{
    public IActionResult Action1()
    {
        TempData["NumberOfUsers"] = 56;
        return View();
    }
}
```

**Action2() Method:**

```
public IActionResult Action2()
{
    return View();
}
```

**View Code (Stage 1):**

```
<h1> TempData test (1/2) </h1>
<p> TempData NumberOfUsers is
    @{
        @((int)(TempData.Peek("NumberOfUsers") ?? 0))
    }
</p>
```

**View Code (Stage 2):**

```
<h1> TempData test (2/2) </h1>
<p> TempData NumberOfUsers is
    @{
        @((int)(TempData["NumberOfUsers"] ?? 0))
    }
</p>
```

**View Code (Stage 3):**

```
<h1> TempData test (2/2) </h1>
<p> TempData NumberOfUsers is
    @{
        @((int)(TempData["NumberOfUsers"] ?? 0))
    }
</p>
```

**Annotations:**

- Stage 1:** "We are reading the element with Peek(). So, the element will be kept till the next request."
- Stage 3:** "This is the second request and TempData["NumberOfUsers"] still exists... However, was marked now to be removed. If we wanted we could mark it not to be removed via Keep() or Peek()."

## WEB | Exercício

- Implemente as operações CRUD respeitantes ao Student. As imagens abaixo apresentam algumas dicas acerca de como popular a lista de Teacher (*tag select* do HTML) com os Teachers existentes na BD.

```
// GET
0 references
public IActionResult Create()
{
    ViewBag.TeacherId = new SelectList(_db.Teachers, "Id", "Name");
    return View();
}

<input asp-for="Fee" class="form-control" />
<span asp-validation-for="Fee" class="text-danger"></span>
</div>
<div class="mb-3">
    <label asp-for="TeacherId" class="form-label"></label>
    <select asp-for="TeacherId" class="form-control" asp-items="ViewBag.TeacherId"></select>
</div>
<button type="submit" class="btn btn-primary">Submit</button>
<a asp-controller="Students" asp-action="Index" class="btn btn-secondary">Back to List</a>
</div>
</form>
```

```
public IActionResult Edit(int? id)
{
    (...)

    // load the teachers list
    ViewBag.TeacherId = new SelectList(_db.Teachers, "Id", "Name", student.TeacherId);

    // show de form
    return View(student);
}
```

```
<input asp-for="Fee" class="form-control" />
<span asp-validation-for="Fee" class="text-danger"></span>
</div>
<div class="mb-3">
    <label asp-for="TeacherId" class="form-label"></label>
    <select asp-for="TeacherId" class="form-control" asp-items="ViewBag.TeacherId"></select>
</div>
<button type="submit" class="btn btn-primary">Submit</button>
<a asp-controller="Students" asp-action="Index" class="btn btn-secondary">Back to List</a>
</div>
</form>
```

- Dropdown in asp.net core using SelectList | Asp.Net Core tutorial  
➤ <https://www.youtube.com/watch?v=MUTUjxXHzzQ>



## LINKS ÚTEIS

- ***Get started with ASP.NET Core MVC***
  - <https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-7.0&tabs=visual-studio>
- ***Using the DropDownList Helper with ASP.NET MVC***
  - <https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/working-with-the-dropdownlist-box-and-jquery/using-the-dropdownlist-helper-with-aspnet-mvc>
- ***Compare Razor Pages to ASP.NET MVC***
  - <https://learn.microsoft.com/en-us/dotnet/architecture/porting-existing-aspnet-apps/comparing-razor-pages-aspnet-mvc>
- ***Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8***
  - <https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-6.0&tabs=visual-studio>

P. PORTO