

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Aplikacje webowe na platformę .NET

W09 – Język C# - atrybuty, silnik Razor,
układ strony, strony częściowe itp.

Syllabus

- Atrybuty
 - Użycie
 - Dostępne
 - Własne atrybuty
 - Atrybuty dla atrybutów
- Model danych MVC
- Model widoku z adnotacjami
- Dodawanie własnego serwisu – przykład
 - Przykładowa implementacja serwisu
- Wstrzykiwanie serwisów poprzez konstruktor – przykład
- Wygenerowanie kontrolera
- Wiązanie danych (data binding) dla kontrolera
- Tworzenie szkieletu widoków dla podstawowych operacji na kolekcji danych
 - Modyfikacja dla wybranego modelu
- widoku
- Pliki widoków i silnik Razor:
 - Pliki wstępne
 - Layout
 - Body
 - Sekcje nazwane
 - Język Razor
 - Widoki częściowe
 - Atrybuty w modelu danych
- Helpery
 - Helpery HTML
 - Tag-helpery
- Wstawianie do kontenera serwisów:
 - `AddSingleton`, `AddScoped`, `AddTransient`

Atrybuty

[`atrybut`]

- Powyżej znajduje się zapis atrybutu.
- Atrybut można stosować do (najczęściej zapisuje się go tuż przed danym elementem):
 - Właściwości
 - Klas
 - Interfejsów
 - Struktur
 - Wyliczeń
 - Delegatów
 - Zdarzeń
 - Metod
 - Konstruktorów,
 - Pól
 - Parametrów
 - Zwracanych wartości (przed metodą z przedrostkiem "`return :`")
 - Podzespołów (inaczej)
 - Parametrów określających typ
 - Modułów (inaczej)

Przykłady użycia atrybutów 1/2

- W EntityFramework (dla baz danych)

```
namespace WebShopEmployeeProvider.Models.Database
{
    // warto stosować anotacje [MaxLength(n)], inaczej w bazie będzie nvarchar(MAX)
    Odwołania: 21
    public class Shop
    {
        // w zasadzie nadmiarowo, Id jest automatycznie traktowane jako klucz
        [Key]
        Odwołania: 11
        public int Id { get; set; }
        [MaxLength(40)]
        Odwołania: 12
        public string Name { get; set; }
        [MaxLength(60)]
        Odwołania: 12
        public string Address { get; set; }
        [Range(20, 10000)]
        Odwołania: 12
        public int AreaM2 { get; set; }
    }
}
```

Przykłady użycia atrybutów 2/2

- Dla opisu akcji kontrolera czy routingu

```
[HttpPost]
[ValidateAntiForgeryToken]
Odwołania: 0
public async Task<IActionResult> Edit(int id, [Bind("Id,Name,Age")] Employee employee)
{
    if (id != employee.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
```

```
[Route("api/[controller]")]
[ApiController]
[EnableCors]
```

1 odwołanie

```
public class StudentController : ControllerBase
{
```

```
    private IRepository repository;
```

Odwołania: 0 | 0 wyjątki

```
    public StudentController(IRepository repo) => repository = repo;
```

```
    [HttpGet]
```

Odwołania: 0 | 0 żądań | 0 wyjątki

```
    public IEnumerable<Student> Get() => repository.Students;
```

```
    [HttpGet("{id}")]
```

1 odwołanie | 0 żądań | 0 wyjątki

```
    public Student Get(int id) => repository[id];
```

```
    [HttpPost]
```

```
    [EnableCors]
```

Odwołania: 0 | 0 żądań | 0 wyjątki

```
    public Student Post([FromBody] Student res) =>
        repository.AddStudent(new Student
        {
            Index = res.Index,
            Name = res.Name
        });
```

Atrybuty - zastosowanie

- Dodają atrybut nie musimy nic dodawać w klasie. Pozwala to np. oddzielić interfejs użytkownika (teksty, typy kontrolek itd.) od informacji biznesowej, ale jednak mieć to w jednym pliku.
- Z tego samego atrybutu można korzystać w wielu innych klasach, każda może go interpretować na swój sposób np.
 - EntityFramework do tworzenia tabel w bazie danych wraz z ograniczeniami
 - Silnik Razor do tworzenia interfejsu i sprawdzenia poprawności danych od użytkownika
- W MVC Core pozwala np. na przekształcanie danych z zapytania POST na parametry metod.
- Itd. itp.

Dostęp do atrybutów z klasy `Type`

- Klasa `Type`, a dokładnie jej metody zwracają klasy pozwalające na dostęp do atrybutów:
 - `PropertyInfo`
 - `MethodInfo`
 - `ParameterInfo`
 - itd.
- W każdej z tych klas istnieją metody:
 - `GetCustomAttributes(bool inherit)`
 - `GetCustomAttributes(Type attributeType, bool inherit)`
- Które zwracają tablicę obiektów `object[]`, który najczęściej rzutuje się na `Attribute[]` lub ten, który został podany w `attributeType`.
- Podobnie `Assembly` posiada analogiczne metody jak wyżej.

Własne atrybuty

- Można tworzyć własne atrybuty
- Powinny się kończyć słowem `Attribute` oraz dziedziczyć po klasie `Attribute`.
- Używając atrybutów można używać albo pełnej nazwy z postfiksem – `Attribute`, albo (najczęściej) bez (.Net sam „doda” ten postfiks).
 - `[NameAttribute("User")]`
 - `[Name("User")]`
- Niektóre atrybuty powinny być tylko przy nazwach klas, inne przy metodach itd.
- Istnieje atrybut (`AttributeUsageAttribute`) do opisu własnej klasy atrybutów, które umożliwiają ograniczenie użycia tworzonych atrybutów.
 - `[AttributeUsage(AttributeTargets.Property)]`
 - `[AttributeUsage(AttributeTargets.Field)]`
 - itd.
- Gdy chcemy użyć atrybut dla kilku miejsc, możemy użyć pionowej kreski „|” jak operator logiczny OR.
 - `[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]`
- Obiekt klasa atrybutów może być w kilku instancjach dla tego samego elementu, wówczas trzeba do klasy tworzonego atrybutu dodać atrybut `AttributeUsage` z parametrem nazwanym `AllowMultiple=true`.
 - `[AttributeUsage(AttributeTargets.Property), AllowMultiple=true]`

Atrybuty - informacje różne

- Ograniczenia:
 - Atrybuty są określane w trakcie kompilacji, zatem wszystkie **parametry** atrybutu muszą być **stałymi**.
 - Mogą być szablony np. „name={ 0 }”;
- Ogólnie temat mechanizmu refleksji i związany z nim temat atrybutów jest dość rozległy, stąd nie są tu przedstawione wszystkie jego elementy.
 - Ostrzeżenia o przestarzałym kodzie
 - Atrybuty dla serializacji
 - Atrybut dla wyłączania warunkowego metod – zamiast `#if/#endif`
- Atrybuty są bardzo szeroko stosowane w ASP .Net Core (i we wcześniejszych wersjach ASP .Net Framework)

Modele danych w MVC

- Modele danych w MVC dzielimy na:

- Model wejściowy

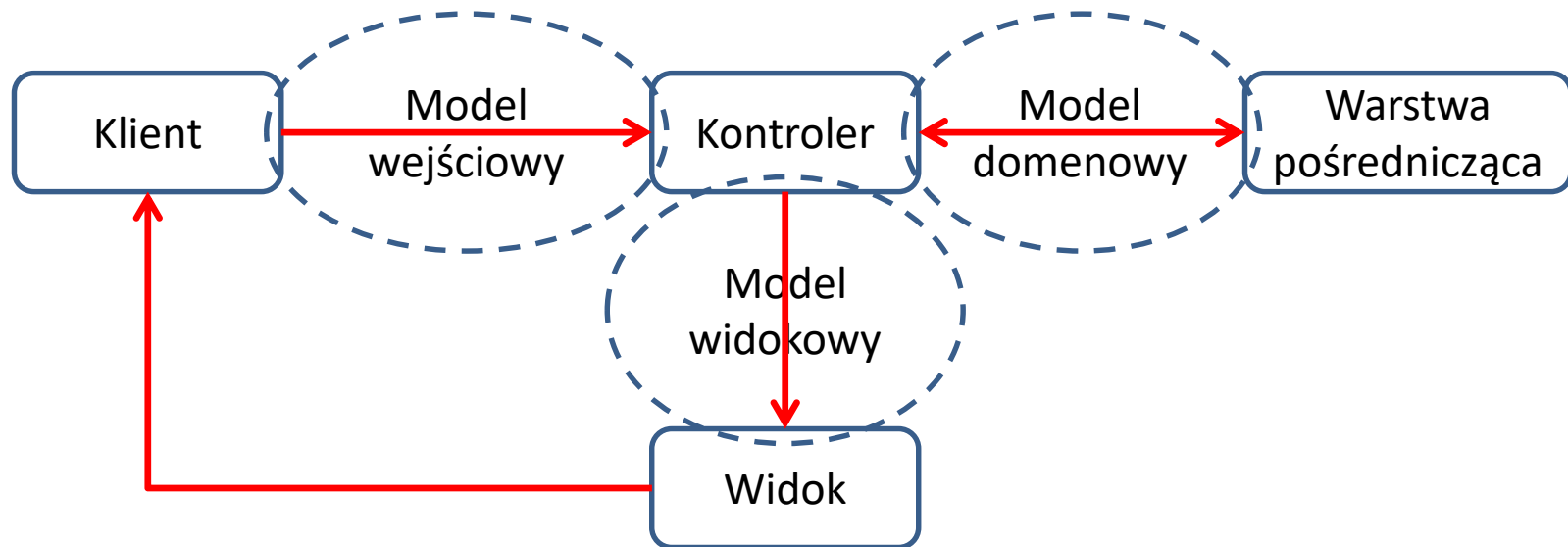
- Między klientem a kontrolerem: reprezentuje zestaw danych przesyłanych przez klienta do kontrolera z wykorzystaniem formularza lub URL.

- Model domenowy

- Między kontrolerem a warstwą pośredniczącą (np. dostęp do bazy danych)

- Model widokowy

- Między kontrolerem a widokiem: Dane oparte o ten model, odpowiednio przetworzone, trafiają do klienta w postaci kodu HTML.



Wstrzykiwanie danych poprzez kontener serwisów

- Dla celów demonstracyjnych stworzony zostanie folder dla widoków modeli (`ViewModels`) a w nim klasa `StudentViewModel`, zawierająca kilka różnych rodzajów właściwości.
 - **Właściwości**, a nie pola
 - Właściwość `Id` będzie **wyznacznikiem** studenta
- Stworzony zostanie folder dla kontekstu danych (`DataContext`), a nim **interfejs** kontekst danych (`IDataContext`), który będzie zawierał metody do manipulowania na danych.
 - Będzie zawierał metody obsługi jednej listy (w praktyce powinien zawierać metody operowania na danych z modelu domenowego)
 - Jako abstrakcja od rzeczywistej reprezentacji kolekcji studentów
- Dla podanego interfejsu dokonana zostanie **implementacja**, która będzie przechowywać dane w pamięci RAM (`MockDataContext`).
 - Dla przyspieszenia demonstracji działania obecna w nim lista studentów zostanie częściowo zapełniona
- **Interfejs** wraz z obecną **implementacją** będzie **zarejestrowany w kontenerze serwisów** oraz **wstrzyknięty do kontrolera**.
 - W tym celu plik `Startup.cs` musi zostać zmodyfikowany (konkretnie metoda `ConfigureServices()`)

Klasy modeli, kontekstu

```
namespace WebAppForRazorDemo.ViewModels
{
    public enum Gender { Female, Male }
    public class StudentViewModel
    {
        public int Id { get; set; }
        [Required]
        [RegularExpression(@"^[0-9]{1,6}$")]
        public int Index { get; set; }
        [Required]
        [MinLength(2, ErrorMessage="Too short name")]
        [Display(Name="Last Name")]
        [MaxLength(20, ErrorMessage="Too long name, do not exceed {1}")]
        public string Name { get; set; }
        public Gender Gender { get; set; }
        public bool Active { get; set; }
        public int DepartmentId { get; set; }
        [DataType(DataType.DateTime)]
        [Required]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        //[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}")]
        public DateTime BirthDate { get; set; }
    }
    // constructors
    ...
}
```

ViewModels/StudentViewModel.cs

```
namespace WebAppForRazorDemo.DataContext
{
    public interface IDataContext
    {
        List<StudentViewModel> GetStudents();
        StudentViewModel GetStudent(int id);
        void AddStudent(StudentViewModel person);
        void RemoveStudent(int id);
        void UpdateStudent(StudentViewModel person);
    }
}
```

DataContext/IDataContext.cs

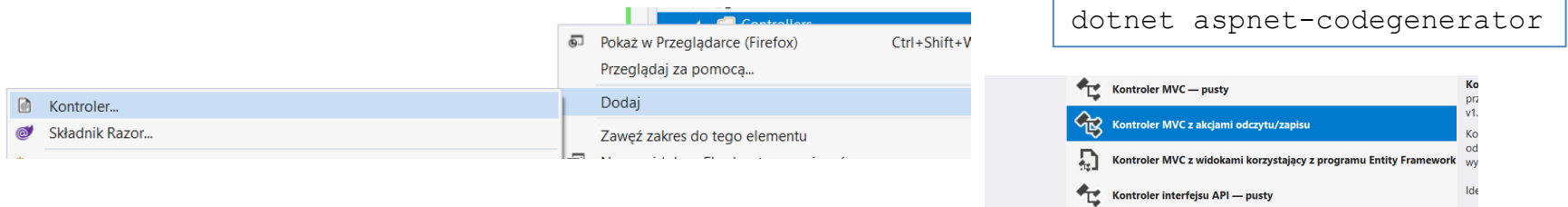
Przykładowa implementacja interfejsu kontekstu

DataContext/MockDataContext.cs

```
public class MockDataContext : IDataContext
{
    List<StudentViewModel> stud = new List<StudentViewModel>() {
        new StudentViewModel(0,123456,"Kowal",Gender.Male,true,2,new DateTime()),
        new StudentViewModel(1,123457,"Newman",Gender.Female,false,1,new DateTime(2000,3,22))
    };
    public void AddStudent(StudentViewModel student)
    {
        int nextNumber=stud.Max(s => s.Id)+1;
        student.Id = nextNumber;
        stud.Add(student);
    }
    public StudentViewModel GetStudent(int id)
    {
        return stud.FirstOrDefault(s => s.Id == id);
    }
    public List<StudentViewModel> GetStudents()
    {
        return stud;
    }
    public void RemoveStudent(int id)
    {
        StudentViewModel studToRemove = stud.FirstOrDefault(s => s.Id == id);
        if(studToRemove != null)
            stud.Remove(studToRemove);
    }
    public void UpdateStudent(StudentViewModel person)
    {
        StudentViewModel studToUpdate = stud.FirstOrDefault(s => s.Id == person.Id);
        stud=stud.Select(s => (s.Id == person.Id) ? person:s).ToList();
    }
}
```

Dodanie serwisu i wstrzyknięcie w konstruktor

- Modyfikacja metody `ConfigureServices()` w `startup.cs`
- Stworzenie kontrolera dla klasy `StudentViewModel`
 - Po stworzeniu modeli widoku w folderze `ViewModels`, PPM na folder `Controllers`, następnie w menu kontekstowym „Add”->”Controller..” a następnie wybieramy „MVC Controller with read/write operation”. Kontroler nazywamy `StudentController`.
- Wytworzy się szkielet kontrolera z metodami `Index`, `Detail`, `Create` itp.
- Dodajemy pole do przechowywania kontekstu i wstrzykujemy zależność od kontekstu w konstruktor kontrolera.



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddSingleton<IDataContext, MockDataContext>();
}
```

Startup.cs

```
public class StudentController : Controller
{
    private IDataContext _dataContext;

    public StudentController(IDataContext dataContext)
    {
        this._dataContext = dataContext;
        ...
    }
}
```

Controllers/StudentController.cs

Modyfikacja kontrolera

- Wytworzony szkielet kontrolera jest ogólny dla akcji Index, Details, Create, Edit i Delete
- Operacje Details, Edit i Delete używają parametru id identyfikatora
 - Zadziała data binding
- Część akcji (Create, Edit, Details) ma dwie wersje (muszą mieć różne nagłówki),
 - jedna dla żądania typu GET – tworzy/pokazuje formularz, który posiada przycisk potwierdzenia wykonania operacji
 - druga dla żądania typu POST – odbiera dane z formularza, wykonuje operację i „przekierowuje” na inną akcję
- Część ma parametr `ICollection collection`.
 - Najprościej zamienić go na parametr `StudentViewModel student`.
 - Zadziała data binding
- Po zadziałaniu data binding dla modelu `ModelState.IsValid` zwraca informację, czy zapisane w adnotacjach ograniczenia są spełnione.

```
public ActionResult Create()
{
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(StudentViewModel student) // change, data binding
{
    try
    {
        if (ModelState.IsValid)
            _dataContext.AddStudent(student);
        return RedirectToAction(nameof(Index));
    }
    catch
    {
        return View();
    }
}
```

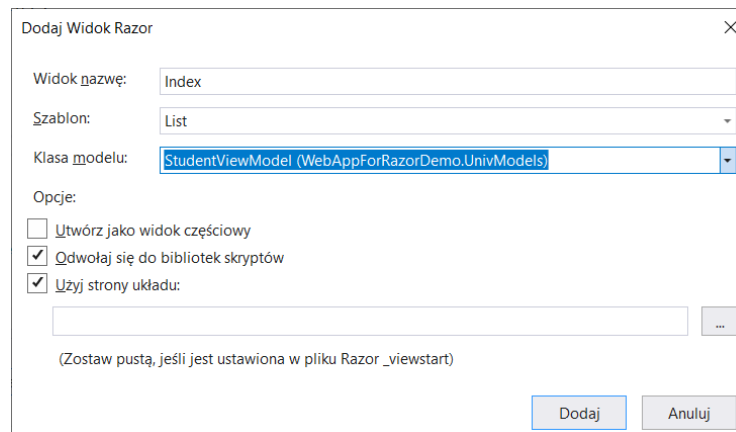
Controllers/StudentController.cs

Data binding

- Technika łącząca źródło danych z odbiorcą danych.
 - W ASP. Net Core wykorzystuje się do tego mechanizm odbicia
 - Nazwa danych/parametru w **źródle** i **odbiorcy** musi być **taka sama**
 - **Źródłem** danych w ASP .NET podczas obsługi żądania HTTP może być:
 - Część adresu URL – wynika z zasady routingu (Route values)
 - `pattern: "{controller=Home}/{action=Index}/{id?}"`
 - `https://localhost:44397/Student/Details/1`
 - Z parametrów po adresie np. w żądaniu typu GET po znaku zapytania '?' znajdują się parametry rozdzielone znakiem '&' będące parami klucz-wartość połączone znakiem równości (spacje i znaki specjalne są odpowiednio kodowane) (Query values)
 - `https://localhost:44397/Student/Details?name=abc&id=1`
 - Z formularza strony, czyli w zapytaniu typu POST, gdzie dane znajdują się w ciele żądania (Form values)
 - Atrybut `name` dla elementu HTML
 - Format jak dla parametrów GET, ale w ciele żądania
 - Odbiorcą danych może być:
 - parametr typu prostego (np. w akcji `Details`)
 - parametr typu złożonego, wtedy następuje dopasowanie **nazw właściwości**. (np. nazwy właściwości dla klasy `StudentViewModel`)
-
- Scenariusz: Reszta analizy na pliku `StudentController.cs`
 - `[HttpPost]` – akcja dla żądania POST
 - Domyślnie lub `[HttpGet]` – akcja dla żądania GET
 - `[ValidateAntiForgeryToken]` – zabezpieczenie przed atakiem Cross-Site Request Forgery (CSRF) – wytłumaczenie na późniejszym wykładzie

Tworzenie szkieletów widoków

- Następnie należy stworzyć widoki. Z analizy kodu wynika, że potrzeba widoków dla akcji: Index, Create, Details, Edit, Delete.
- Aby je stworzyć należy ustawić myszkę na metodzie kontrolera (np. Details) PPM, wybrać „Add View/Dodaj widok”.
- Wybrać „Widok Razor”
- W oknie „Add View” zostawić nazwę domyślną (np. „Details”) lub ją zmienić, jako Template wybrać odpowiedni szablon (w tym przykładzie „Details”), w Model class wpisać StudentViewModel (WebAppForRazorDemo.Models) i wcisnąć przycisk „Add”.
- W folderze Views/Student pojawi się plik Details.cshtml.
- Analogicznie wytworzyć pliki Create.cshtml, Index.cshtml, Edit.cshtml oraz Delete.cshtml
- Tak stworzone widoki są przygotowane pod działania na bazie danych, więc trzeba je zmodyfikować.



Dodaj Widok Razor

Widok nazwę: Index

Szablon: List

Klasa modelu: StudentViewModel (WebAppForRazorDemo.UnivModels)

Opcje:

☐ Utwórz jako widok częściowy

☒ Odwołaj się do bibliotek skryptów

☒ Użyj strony układu:

(Zostaw pustą, jeśli jest ustawiona w pliku Razor _viewstart)

Dodaj Anuluj

dotnet aspnet-codegenerator

Modyfikacja widoku Person/Index.cshtml

- W linkach generowanych przez `@Html.ActionLink` należy dodać `id`, które jest w komentarzu, w poprawny sposób.
- Zamienić

View/Person/Index.cshtml

```
<td>
    @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
    @Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ }) |
    @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
</td>
```

- na

```
<td>
    @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
    @Html.ActionLink("Details", "Details", new { id = item.Id }) |
    @Html.ActionLink("Delete", "Delete", new { id = item.Id })
</td>
```

- Podobnie w widoku `Details.cshtml` zmodyfikować skomentowane fragmenty np.:

```
@Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
```

- na

View/Person/Details.cshtml

```
@Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
```

- Identyfikator ma być tworzony, a nie wpisywany/modyfikowany, więc usunięta zostanie część odpowiedzialna z jego zmianę w widoku `Create.cshtml`

```
<div class="form-group">
    <label asp-for="Id" class="control-label"></label>
    <input asp-for="Id" class="form-control" />
    <span asp-validation-for="Id" class="text-danger"></span>
</div>
```

Efekt działania

- <https://localhost:44397/Student>

The screenshot shows a web browser at <https://localhost:44397/Student>. The page title is "WebAppForRazorDemo" with navigation links for "Home", "Privacy", and "Students". The main heading is "Index". A link labeled "Create New" is highlighted with a red dashed box. Below it is a table of students. A blue arrow points from the "Create New" link to a text box containing the URL <https://localhost:44397/Student/Create>.

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete

Tworzenie nowego studenta

- Działają walidatory z adnotacji
- Poprawnie dodaje do listy studentów

Create
StudentViewModel

Index

Last Name

Gender
Please select

☐ Active

DepartmentId

BirthDate
dd.mm.rrrr, --:--

Create

[Back to List](#)

Create
StudentViewModel

- Please enter a valid number.
- The Last Name field is required.
- Please enter a valid number.
- The BirthDate field is required.

Index
aaa

Please enter a valid number.

Last Name

The Last Name field is required.

Gender
Female

☐ Active

DepartmentId
www

Please enter a valid number.

BirthDate
dd.10.rrrr, --:--

The BirthDate field is required.

Create

[Back to List](#)

Create
StudentViewModel

Index
222222

Last Name
Konieczny

Gender
Male

☒ Active

DepartmentId
3

BirthDate
10.10.1999, 00:00

Create

[Back to List](#)

Index

[Create New](#)

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete
2	222222	Konieczny	Male	<input checked="" type="checkbox"/>	3	1999-10-10 00:00:00	Edit Details Delete

Modyfikacja studenta

Index

[Create New](#)

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete
2	222222	Konieczny	Male	<input checked="" type="checkbox"/>	3	1999-10-10 00:00:00	Edit Details Delete

Edit

StudentViewModel

Id
2

Index
333333

Last Name
Konieczny

Gender
Male

☐ Active

DepartmentId
3

BirthDate
10.10.1999, 00:00

[Save](#)

[Back to List](#)

<https://localhost:44397/Student/Edit/2>

Index

[Create New](#)

	Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete	
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete	
2	333333	Konieczny	Male	<input type="checkbox"/>	3	1999-10-10 00:00:00	Edit Details Delete	

Usuwanie studenta

Index

[Create New](#)

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete
2	333333	Konieczny	Male	<input type="checkbox"/>	3	1999-10-10 00:00:00	Edit Details Delete

<https://localhost:44397/Student/Delete/2>

Delete

Are you sure you want to delete this?
StudentViewModel

Id	2
Index	333333
Last Name	Konieczny
Gender	Male
Active	<input type="checkbox"/>
DepartmentId	3
BirthDate	1999-10-10 00:00:00

[Delete](#) | [Back to List](#)

Index

[Create New](#)

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete

Silnik Razor

Tworząc stronę WWW za pomocą pliku z widokiem (np. `/Views/Student/Index`):

- Na początku ładowane są do analizy i kompilacji pliki:
 - `_ViewImports.cshtml` – powinien zawierać importowanie, poprzez `using` zespoły potrzebne do działania wszystkim widokom i ewentualnie używane tag-helpery
 - `_ViewStart.cshtml` – zawiera kod wykonywane dla każdej strony, najczęściej jest tam ustawienie domyślnego układu strony (`Shared/Layout.cshtml`)
- Razor wykonuje wiązanie danych przekazanych przez kontroler z właściwością `Model`. Poprzez pierwszą instrukcję w pliku:
 - `@model IEnumerable<WebAppForRazorDemo.ViewModels.StudentViewModel>`
- W pliku z widokiem, podczas tworzenia strony HTML na jego podstawie, najpierw analizowany jest jego układ (`Layout`).
 - Domyślny lub wybrany na początku kodu danej strony
- Plik z widokiem jest w „wypełnieniu” układu strony w miejscu `@RenderBody()`
- Plik widoku może posiadać oprócz ciała jeszcze sekcje nazwane, którymi wypełnia się inne miejsca w układzie strony
- W razie istnienia pobierane są i „wklejane” widoki częściowe.

`Views/_ViewStart.cshtml`

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

Początek pliku View/Shared/_Layout.cshtml

- Poprawny początek pliku HTML5, elementy:

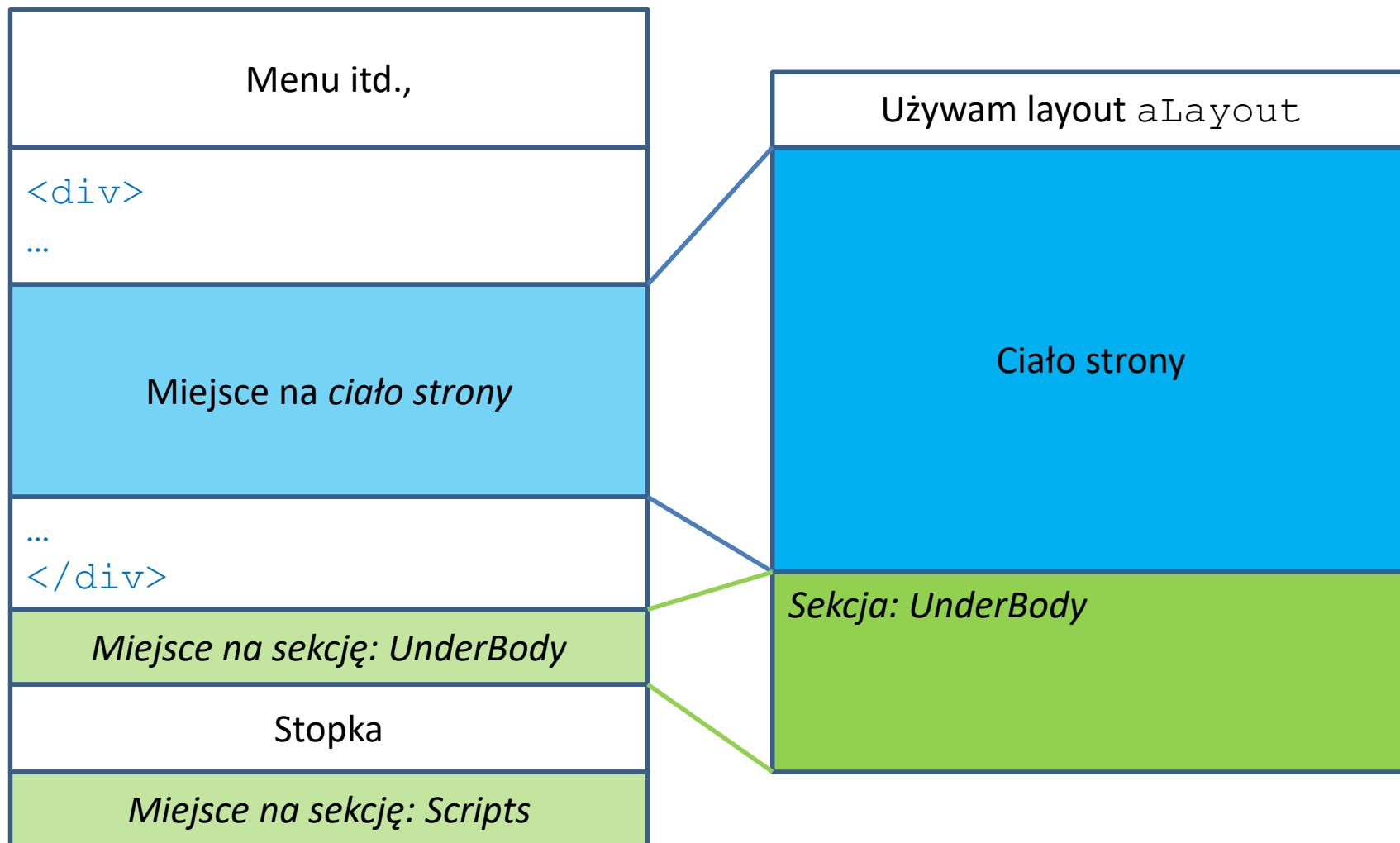
- <!DOCTYPE>
- <html>
- <head>
- <body>

```
_Layout.cshtml
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>@ViewData["Title"] - WebAppForRazorDemo</title>
7      <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
8      <link rel="stylesheet" href="~/css/site.css" />
9  </head>
10 <body>
11     <header>
12         <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
13             <div class="container">
14                 <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">WebAppForRazorDemo</a>
15                 <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-c
16                     aria-expanded="false" aria-label="Toggle navigation">
17                     <span class="navbar-toggler-icon"></span>
18                 </button>
19                 <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
20                     <ul class="navbar-nav flex-grow-1">
21                         <li class="nav-item">
22                             <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
23                         </li>
```


Zależności

aLayout

aView



Sekcje nazwane - przykład

- W pliku definiującym układ można używać **sekcje nazwana**. Sekcje nazwane pozwalają na wstawienie **miejsc do wypełnienia w układzie strony (layout)**. Oprócz nich zawsze jest **główne ciało strony**.
- „Wstawienie” oznacza renderowanie (na HTML5), stąd jest to wykonywane poprzez wywołanie metody `RenderBody()` lub `RenderSection()` dostępne w silniku Razor.

Views/Shared/_Layout.cshtml

```
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>
</div>

@RenderSection("UnderBody", required: false)

<footer class="border-top footer text-muted">
  <div class="container">
    &copy; 2020 - WebAppForRazorDemo - <a asp
  </div>
</footer>
<script src="~/lib/jquery/dist/jquery.min.js"></s
<script src="~/lib/bootstrap/dist/js/bootstrap.bu
<script src="~/js/site.js" asp-append-version="tr
  @RenderSection("Scripts", required: false)
</body>
```

Sekcje nazwane - wyjaśnienie

- Główne ciało widoku będzie w miejscu `@RenderBody()`
- Pozostałe sekcje będą w miejscach zależnych od nazwy sekcji, np. `@RenderSection("UnderBody", required: false)`
- Parametr „`required: false`” oznacza, że dana sekcja nie musi wystąpić na stronie. Gdyby ustawić ten parametr na **true**, brak sekcji będzie generować wyjątek.
- Przykładowe użycie sekcji nazwanych w widoku:

```
<td>
    @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
    @Html.ActionLink("Details", "Details", new { id=item.Id }) |
    @Html.ActionLink("Delete", "Delete", new { id=item.Id })
</td>
</tr>
</tbody>
</table>
```

Views/Student/Index.cshtml

```
@section UnderBody
{
    
}
```

Index

[Create New](#)

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Delete



End of list



Język silnika Razor

- Język ten pozwala na mieszanie kodu HTML z kodem silnika Razor.
- Pełny opis można znaleźć po adresem:
<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor>
- Jeśli w pliku widoku pojawi się znak '@', następuje przejście do kodu Razor/C#. Jeśli po tym znaku występuje bezpośrednio słowo kluczowe Razora, wykonuje się pewna akcje tego silnika, w p.p. interpretacja kodu C#.
- Jeśli w tekście HTML potrzebujemy znaku '@', piszemy go podwójnie „@@”
 - Wyjątek – atrybuty HTML związane z adresem email i ich zawartość będące adresem, nie wymagają podwójnego znaku '@', np.:
`Support@contoso.com`
- 1) Wyrażenia bezpośrednie Razora:
 - Bezpośrednio po '@' piszemy wyrażenie proste C# kończące się spacją (białym znakiem) lub innym znakiem nie pasującym do wyrażenia.
 - Wyjątek: Po `await` po spacji są argumenty np.:
`<p>@await DoSomething("hello", "world")</p>`

Razor – wyr. pośrednie i bloki kodu

- 2) Wyrażenia pośrednie: @ (...)
 - Wyrażenie pomiędzy nawiasami będzie obliczone i jego wynik wstawiony do HTML jako tekst.
- 3) Blok kodu: @ { ... }
 - Domyślnym językiem w takim bloku jest C#
 - Jeśli na początku linii pojawi się znacznik HTML, nastąpi przełączenie na język HTML dla tej linii.
 - Jeśli chcemy wymusić przełączenie bez znacznika HTML, linię należy zacząć od „@ :”
 - Kolejne bloki kodu są jakby fragmentem jednej funkcji, zatem zmienne stworzone we wcześniejszych fragmentach są dostępne w kolejnych fragmentach.

Razor - przykłady

- Widok do testów języka Razor.
 - Kolejne listingi również pochodzą z pliku `Views/Home/Test.cshtml`.
- Wyrażenie bezpośrednie **nie zakańcza się** średnikiem.
 - Średnik zostanie wtedy elementem dokumentu HTML.

`Views/Home/Test.cshtml`

```
@{
    ViewBag.Title = „Test”;
}

@using WebAppForRazorDemo.ViewModels
...

@{
    var students = new StudentViewModel[]
    {
        new StudentViewModel(0,123456,"Kowal",Gender.Male,true,2,new DateTime()),
        new StudentViewModel(1,123457,"Newman",Gender.Female,false,1,new DateTime(2000,3,22))
    };
}
```

Razor przykłady

Wyrażenie bezpośrednie

```
<p>@DateTime.Now</p>  
<p>@DateTime.IsLeapYear(2016)</p>
```

```
<p>@await DoSomething("hello", "world")</p>
```

WebAppForRazorDemo Home

2021-12-08 15:48:16

True

Last week this time: 2021-12-01 15:48:16

Wyrażenie pośrednie

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Gdy tekst byłby zinterpretowany jako adres w wyrażenie bezpośrednim, należy użyć wyrażenie pośredniego (jest to też przykład na blok kodu):

```
@{  
    var joe = new StudentViewModel(0, 123456, "Joe", Gender.Male, true, 2, new DateTime());  
}  
<p>Name@(joe.Name)</p>
```

Last week this time: 2021-12-01 15:48:16

NameJoe

Test

Automatyczna zmiana z kodu C# na kod HTML

```
@{  
    var inCSharp = true;  
    <p>Now in HTML, was in C# @inCSharp</p>  
}
```

TEST

Now in HTML, was in C# True

Using @if

```
@for (var y = 0; y < students.Length; y++)  
{  
    var student = students[y];  
    <p> Name: @student.Name</p>  
}
```

Now in HTML, was in C# True

Name: Kowal

Name: Newman

Wymuszona zmiana z kodu C# na kod HTML (skomplikowane!)

```
@for (var x = 0; x < students.Length; x++)  
{  
    var student = students[x];  
    @:Name: @student.Name  
}
```

Name: Kowal Name: Newman

```
@for (var x = 0; x < students.Length; x++)  
{  
    <p>  
        var student = students[x];  
        @:Name: @student.Name  
    </p>  
}
```

Compilation error

Razor – struktury kontrolne

- @if, else if, else
- @switch, case, default

Views/Test.cshtml

```
<h3>Using @@if</h3>
@{ var number = 11111;}
@if (number % 2 == 0)
{
    <p>The number was even</p>
}
else if (number >= 1337)
{
    <p>The number is large.</p>
}
else
{
    <p>The number was not large and is odd.</p>
}
<h3>Using @@switch</h3>
@switch (number)
{
    case 1:
        <p>The number is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number was not 1 or 1337.</p>
        break;
}
```

Using @if

The number is large.

Using @switch

Your number was not 1 or 1337.

Razor – struktury kontrolne

- **@for**
- **@foreach**

Views/Test.cshtml

```
<h3>Using @@for</h3>

@for (var i = 0; i < students.Length; i++)
{
    var stud = students[i];
    <p>Name: @stud.Name</p>
    <p>Index: @stud.Index</p>
    <hr />
}

<h3>Using @@foreach</h3>
@foreach (var person in students)
{
    <p>Name: @person.Name</p>
    <p>Index: @person.Index</p>
    <hr />
}
```

Using @for

Name: Kowal

Index: 123456

Name: Newman

Index: 123457

Using @foreach

Name: Kowal

Index: 123456

Name: Newman

Index: 123457

Razor – struktury kontrolne

- @while
- @do while

Views/Test.cshtml

```
<h3>Using @@while</h3>

@{ var i2 = 0; }
@while (i2 < students.Length)
{
    var person = students[i2];
    <p>Name: @person.Name</p>
    <p>Index: @person.Index</p>

    i2++;
    <hr />
}

<h3>Using @@do while</h3>
@{ var i3 = 0; }
@do
{
    var person = students[i3];
    <p>Name: @person.Name</p>
    <p>Index: @person.Index</p>

    i3++;
} while (i3 < students.Length);
```

Using @while

Name: Kowal

Index: 123456

Name: Newman

Index: 123457

Using @do while

Name: Kowal

Index: 123456

Name: Newman

Index: 123457

Razor – inne elementy

- @using (x2)
- @* *@ – komentarz

Views/Test.cshtml

```
<h3>Using @@using</h3>

@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>Current folder is: @dir</p>

@using (Html.BeginForm())
{
    <div> email: <input type="email" id="Email" name="Email" value="" />
    <button type="submit"> Register </button> </div>
}

<h3>Using @@* - begin</h3>

@*
    very long comment
    many lines
*@

<h3>Using *@@ - end</h3>
```

Using @using

Current folder is: C:\Users\dariu\source\repos\WebAppForRazorDemo\WebAppForRazorDemo

email:

Using @* - begin

Using *@@ - end

Razor – model, section

- `@model` – ustala jaka klasa modelu danych dla widoku będzie używana. Obiekt tego typu dostępny jest poprzez właściwość `Model`. W helperach Razora właściwość ta jest najczęściej pobierana jako parametr dla wyrażeń lambda.
 - Poniżej przykład dla `View/Student/Index`
- Model ten jest przekazywany przez metodę `View(model)` w kontrolerze.
 - Poniżej przykład dla `StudentController`.

```
Index.cshtml  + X
1  @model IEnumerable<WebAppForRazorDemo.ViewModels.StudentViewModel>
2
3  @{
4      ViewData["Title"] = "Index";
5  }
6
7  <table border="1">
8      <tr>
9          <th>Id</th>
10         <th>Name</th>
11     </tr>
12     <tbody>
13         @foreach (var item in Model)
14         {
15             <tr>
16                 <td>
17                     @Html.DisplayFor(modelItem => item.Id)
18                 </td>
19                 <td>
20                     @Html.DisplayFor(modelItem => item.Name)
21                 </td>
22             </tr>
23         }
24     </tbody>
25 </table>
```

```
// GET: StudentController
3 references
public ActionResult Index()
{
    return View(_dataContext.GetStudents()); // added parameter
}
```

Razor inne elementy

- `@section` – było wcześniej wytłumaczone

Mniej istotne (bardziej zaawansowane):

- `@inherit` – strona WWW będzie dziedziczyć po innej klasie C# niż standardowo używa Razor (bardziej specjalizowanej)
- `@inject` – wykorzystanie kontenera serwisów do wstrzyknięcia wartości
- `@function` – dynamiczne dodanie nowej funkcji C# do klasy tworzonej dla każdego widoku.

Widoki częściowe

- **Widoki częściowe** (partial views) znajdują się najczęściej w folderze `Views/Shared` i ich nazwy zaczynają się **od znaku podkreślenia**.
- Oznaczają fragment strony, który dostaje najczęściej jakiś model danych i generuje fragment kodu tej strony (**bez korzystania z szablonu strony!**).
- Aby skorzystać z takiego widoku częściowego należy wykorzystać helper `@Html.Partial(<nazwaStrony>, <daneModelu>)`, np.:
`@Html.Partial("__Student", Model[i])`
- Istnieje podobna metoda `Html.RenderPartial()` z takimi samymi argumentami. Różnica:
 - `Html.Partial()` zwraca `MvcHtmlString`, który następnie przekierowujemy na strumień wyjściowy. Może tym stringiem również manipulować.
 - `Html.RenderPartial()` jest typu `void`, od razu wysyła dane na strumień wyjściowy, przez to jest szybsze, jednak nie można już nic w tym stringu zmienić oraz trzeba ją wywołać jako blok kodu, czyli:
`@{Html.RenderPartial(...);}`
- Od wersji Core 2.1 wskazane jest użycie metod asynchronicznych i czekanie na ich wykonanie:
 - **await** `Html.RenderPartialAsync(...)`;
 - Wówczas dodatkowym parametrem może być słownik `ViewData` wysyłany do widoku częściowego.

Widoki częściowe - przykład

```
public ActionResult AnotherIndex()  
{  
    return View(persons);  
}
```

Controllers/StudentController.cs

Views/Student/AnotherIndex.cshtml

```
@model IEnumerable<WebLayout.ViewModels.StudentViewModel>  
@{  
    ViewBag.Title = „AnotherIndex”;  
}  
<h2>@ViewBag.Title</h2>  
<table>  
    @foreach (var item in Model)  
    {  
        //Html.RenderPartial("_Student", item);  
        await Html.RenderPartialAsync("_Student", item);  
    }  
</table>
```

Views/Shared/_Student.cshtml

```
@model WebAppForRazorDemo.ViewModels.StudentViewModel  
<tr>  
    <td> @Model.Name</td>  
    <td> @Model.Index</td>  
</tr>
```


Atrybuty w modelu danych

- `using System.ComponentModel.DataAnnotations;`
- Można używać kilka adnotacji do jednej właściwości

```
namespace WebAppForRazorDemo.ViewModels
{
    public enum Gender { Female, Male }
    public class StudentViewModel
    {
        public int Id { get; set; }
        [Required]
        [RegularExpression(@"^[0-9]{1,6}$")]
        public int Index { get; set; }
        [Required]
        [MinLength(2, ErrorMessage="To short name")]
        [Display(Name="Last Name")]
        [MaxLength(20,ErrorMessage =" To long name, do not exceed {1}")]
        public string Name { get; set; }
        public Gender Gender { get; set; }
        public bool Active { get; set; }
        public int DepartmentId { get; set; }
        [DataType(DataType.DateTime)]
        [Required]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        //[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}")]
        public DateTime BirthDate { get; set; }
    }
    // constructors
    ...
}
```

Adnotacje

- Istnieje wiele adnotacji nakładające ograniczenia dla właściwości. Analizowane są przez silnik Razor (oraz `ModelState.IsValid`):
 - Pole wymagane (Required)
 - Minimalna długość
 - Maksymalna długość
 - Zakres (Range)
 - Porównanie dwóch pól (dla powtarzania hasła/emaila)
 - Wyrażenie regularne
 - Inne
- Istnieją adnotacje dla zmiany formatu danych lub etykiety przyporządkowanej do właściwości (standardowy tekst w etykiecie to nazwa właściwości).
 - `[Display(Name="Last Name")]`
 - `[DataType(DataType.DateTime)]`
 - `[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]`
- Standardowe komunikaty o błędnych danych mogą być nieczytelne dla użytkownika strony WWW
 - Np. dla właściwości `Index`.
- Adnotacje ograniczające mogą posiadać parametry dodatkowe np. treść komunikatu w wypadku błędu
 - `[MaxLength(20, ErrorMessage = " To long name, do not exceed {1}")]`, gdzie {0} to nazwa etykiety, {1} to wartość tej adnotacji (20), {2} i kolejne to ewentualne dodatkowe parametry (wyrażenia stałe)
- W widoku CSHTML odpowiedni kod korzysta z tych adnotacji do wstawienia poprawnej etykiety, kontrolki do wprowadzania danych oraz do informowaniu o niepoprawnych danych.

Walidacja modelu

- W kodzie HTML używane są atrybuty dla jQuery (ogólnie JavaScript)
 - Dla programisty w ASP jest to dość nieistotne, bo generowane automatycznie

Create

StudentViewModel

- The field Index must match the regular expression '^[0-9]{1,6}\$'.
- To short name
- Please enter a valid number.
- The BirthDate field is required.

Index

22222222

The field Index must match the regular expression '^[0-9]{1,6}\$'.

Last Name

a

To short name

Gender

Female

☐ Active

DepartmentId

asas

Please enter a valid number.

BirthDate

The BirthDate field is required.

Create

```
<form asp-action="Create">
  <div asp-validation-summary="All" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="Index" class="control-label"></label>
    <input asp-for="Index" class="form-control" />
    <span asp-validation-for="Index" class="text-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="Name" class="control-label"></label>
    <input asp-for="Name" class="form-control" />
    <span asp-validation-for="Name" class="text-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="Gender" class="control-label"></label>
    <select asp-for="Gender" class="custom-select" asp-items="Html.GetEnumSelectList<Gender>()">
      <option value="">Please select</option>
    </select>
    <span asp-validation-for="Gender" class="text-danger"></span>
  </div>
  <div class="form-group form-check">
    <label class="form-check-label">
      <input class="form-check-input" asp-for="Active" /> @Html.DisplayNameFor(model => model.Active)
    </label>
  </div>
  <div class="form-group">
    <label asp-for="DepartmentId" class="control-label"></label>
    <input asp-for="DepartmentId" class="form-control" />
    <span asp-validation-for="DepartmentId" class="text-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="BirthDate" class="control-label"></label>
    <input asp-for="BirthDate" class="form-control" />
    <span asp-validation-for="BirthDate" class="text-danger"></span>
  </div>
  <div class="form-group">
    <input type="submit" value="Create" class="btn btn-primary" />
  </div>
</form>
```

Helpers - ogólnie

- Helpers to różne sposoby zapisu, które powodują, że zamiast pisać bezpośrednio kod HTML można stosować zapis alternatywny, który pozwoli:
 - Na stosowanie Intellisense
 - Uwidoczni powiązanie z modelami widoków
 - Rozpozna błędy powiązań, które w kodzie HTML są po prostu ciągami w cudzysłowach
 - Pozwoli na łatwą modyfikację w przyszłości
- W ASP .Net istnieją różne rodzaje helperów:
 - HTML-Helpers
 - Tag-Helpers
 - inne

Razor - Helperzy HTML - ogólnie

- Helperzy HTML są to **klasy C#**, których zadaniem jest wygenerowanie kodu HTML, który będzie wstawiony w miejscu helpera.
 - Tworzony będzie **cały znacznik**
- Zalecane nazewnictwo:
 - Wywołanie na stronie .cshtml: `@Html.<HelperName>(parameters)`
 - Klasa dla Helpera: `<HelperName>Helper`
 - Musi być publiczna statyczna metoda rozszerzająca w tej klasie zwracająca string, o nagłówku: **`public static string <HelperName>(this HtmlHelper helper, <otherParams>)`**
 - Może być kilka metod różniących się parametrami.
- Przykład:
 - Wywołanie: `@Html.SubmitButton("Create Customer")`
 - Klasa: `SubmitButtonHelper`
 - Metoda/y w powyższej klasie:
`public static string SubmitButton(this HtmlHelper helper, string buttonText)`
- Jest wiele gotowych helperów, można też tworzyć własne zgodnie z powyższym schematem.
- Można też tworzyć helpery nie trzymając się powyższego schematu.
 - Helpery można pisać też na stronie CSHTML (w Razorze).

Wbudowane HTMLhelpery

- **Wbudowane helpery:**
 - `Html.Label`
 - `Html.Editor`
 - `Html.BeginForm`
 - `Html.EndForm`
 - `Html.TextBox`
 - `Html.TextArea`
 - `Html.Password`
 - `Html.Hidden`
 - `Html.CheckBox`
 - `Html.RadioButton`
 - `Html.DropDownList`
 - `Html.ListBox`
 - `Html.ValidationSummary`
 - `Html.ActionLink`
 - `Itd.`
- **Narzędzie Intellisense podpowiada możliwe zestawy parametrów dla tych helperów.**

Wbudowane helpery - przykład

- Helpery te mają odpowiednie parametry zależne od kontrolki, dla której zostały przygotowane.
- Dodatkowo poprzez mechanizm dynamicznych typów można do nich wysłać dodatkowe dane, które zostaną dopisane do odpowiednich kontrolek (wiedza nt. kontrolek HTML i np. Bootstrap'a).

Views/Home/TestHtmlHelper.cshtml

```
@Html.Label("exampleLabel", " Example label:")
<hr />
@Html.Label("lblta", "TextArea:")
@Html.TextArea("textArea1", "initial text", 4, 30, null)
<hr />
@Html.Label("editor1", "Editor:")
@Html.Editor("editor1")
<hr />
@Html.Label("textBox1", "TextBox:")
@Html.TextBox("textBox1", "text")
<hr />
@Html.Label(" pass1 ", "Password:")
@Html.Password("pass1")
<hr />
@Html.Label("checkBox1", "CheckBox:")
@Html.CheckBox("checkBox1",true)
<hr />
```

Wbudowane helpery - przykład

```
@Html.Label("lblradbut", "RadioButton:")
<br />
@Html.Label("radioButton1", "female:")
@Html.RadioButton("radioButton1", "female", false);
@Html.Label("radioButton1", "male:")
@Html.RadioButton("radioButton2", "male", false);
<hr />
@Html.Label("DropDown1", "DropDownList:")
@Html.DropDownList("DropDown1", new SelectList(
    new List<Object>{
        new { value = 0 , text = "Red" },
        new { value = 1 , text = "Blue" },
        new { value = 2 , text = "Green" }
    },
    "value",
    "text",
    2))

<hr />
@Html.Label("listBox1", "ListBox:")
@Html.ListBox("listBox1", new SelectList(
    new List<Object>{
        new { value = 0 , text = "Red" },
        new { value = 1 , text = "Blue" },
        new { value = 2 , text = "Green" }
    },
    "value",
    "text",
    2))

<hr />
```

Test HTML helpers

Example label:

TextArea:

Editor:

TextBox:

Password:

CheckBox: ☒

RadioButton:
female: ☐; male: ☐

DropDownList:

ListBox:

Red

Blue

Green

Views/Home/TestHtmlHelper.cshtml

Wbudowane helpery dla typów silnych

- Gdy mamy dane konkretnego typu (np. `int`), wiadomo, że nie wszystkie ciągi znaków są poprawne, gdy taką daną wpisujemy do formularza. Są przygotowane specjalne helpery do działania dla takich sytuacji (czyli sprawdzające poprawność podawanych danych)
 - `Html.EditorFor`
 - `Html.LabelFor`
 - `Html.TextBoxFor`
 - `Html.TextAreaFor`
 - `Html.PasswordFor`
 - `Html.HiddenFor`
 - `Html.CheckBoxFor`
 - `Html.RadioButtonFor`
 - `Html.DropDownListFor`
 - `Html.ListBoxFor`
- Pierwszym argumentem tych helperów jest wyrażenie lambda (za pomocą którego przekazany zostanie typ), kolejne zależą od helpera.
- Wada HTML-helperów: ponieważ helper zamienia się na **cały znacznik HTML** dodatkowe atrybuty znacznika trzeba przesłać przez obiekt typu anonimowego z polami jak nazwy atrybutów np.:
`Html.TextBoxFor(m=>m.Name, new { @class = "form-control" })`
 - Miały odciążać od pamiętania składni HTML, ale nie do końca spełniają tę rolę

Tag helpers 1/3

- Nowsza technologia , podejście odwrotne od helperów html – używajmy jak atrybutów znaczników HTML
 - Zapis jak atrybut HTML
 - Zamienia na inny, poprawny zapis HTML, będący **fragmentem zapisu** znacznika
- Najlepiej dodać możliwość korzystania z tag-helperów w pliku `_ViewImports.cshtml` poprzez (dla projektu MVC jest robione automatycznie):
 - `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`
- Oznaczenia ASP wewnątrz tagów HTML mają początek „**asp-**”
- Ogólne:
 - `asp-for` – dla której właściwości z modelu jest dana kontrolka

Tag helpers 2/3

- Dla adresowania:
 - `asp-controller` – który kontroler ma wykonać akcję
 - `asp-action` – którą akcję
 - `asp-route` – nazwa reguły routingu, która ma zostać wykorzystana
 - `asp-route-{valueName}` – który parametr z reguły routingu ustawić na jaką wartość
 - `asp-protocol` – który protokół użyć
 - i inne
- Dla widoków częściowych:
 - `name`
 - `for`
 - `model`
 - i inne
- Dla walidacji:
 - `asp-validation-for` – dla której właściwości z modelu umieścić informację o niepoprawnej walidacji
 - `asp-validation-summary` – podsumowanie których walidacji wstawić (All/ModelOnly/None)
 - i inne
- i inne

Tag helpers 3/3

- Klasa uruchamiania przez silnik Razor do interpretacji tag-helpera otrzymuje informację zarówno o wartości za znakiem '=' ale także:
 - O wartości {value} dla `asp-route`
 - W jakim znaczniku tag helper został użyty
 - Jakie są inne atrybuty tego znacznika
 - W jakim kontekście (ścieżka/fragment ścieżki) w modelu DOM został użyty
 - itp.
- Można tworzyć własne tag helpers. Istnieją atrybuty pozwalające ograniczyć używanie tag helperów tylko do wybranych poprawnych wystąpień na podstawie w/w informacji.
- Dla części tag helperów (np. dla `asp-for`, `asp-validation-summary`) działa Intellisense, podpowiadający poprawne wartości (albo na podstawie modelu strony CSHTML, albo możliwe stałe)
- Nie wszystkie HTML-Helpers mają swoje odpowiedniki w tag-helperach, stąd w kodzie CSHTML pojawiają się obydwa rodzaje helperów

Dlaczego helpery?

- Dla linków - tworzone są na podstawie zasad routingu
 - Zmiana zasad routingu wygeneruje nowy link
- Dla kontrolek dla modelu – podpowiada poprawne nazwy
- Tag helpery mogą generować inną treść w zależności od kontekstu

```
[Route("newDetails/{id}", Name = "newDetailsRoute")]  
// GET: StudentController/Details/5  
Odwolania: 0  
public ActionResult Details(int id)  
{  
    return View(_dataContext.GetStudent(id)); // added parameter  
}
```

Index

Create New

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Details (HC) Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Details (HC) Delete

<https://localhost:44397/newDetails/0>

Views/Student/Index.cshtml

```
<td>  
    @Html.ActionLink("Edit", "Edit", new { id = item.Id }) |  
    @Html.ActionLink("Details", "Details", new { id = item.Id }) |  
    <a href="/Student/Details/@item.Id">Details (HC)</a> |  
    @Html.ActionLink("Delete", "Delete", new { id = item.Id })  
</td>
```

Index

Create New

Id	Index	Last Name	Gender	Active	DepartmentId	BirthDate	
0	123456	Kowal	Male	<input checked="" type="checkbox"/>	2	0001-01-01 00:00:00	Edit Details Details (HC) Delete
1	123457	Newman	Female	<input type="checkbox"/>	1	2000-03-22 00:00:00	Edit Details Details (HC) Delete

<https://localhost:44397/Student/Details/1>

Tag helper z wykorzystaniem zasady routingu:

```
<a asp-route="newDetailsRoute" asp-route-id="@item.Id"> Details(asp-)</a>
```

Wstawianie do kontenera serwisów

- Wstawianie implementacji interfejsu (klasy) do kontenera serwisu można wykonać na 3 sposoby. Różnią się one sposobem tworzenia instancji klasy.
 - `AddSingleton<IServiceA, ClassServiceA>()`
 - `AddScoped<IServiceA, ClassServiceA>()`
 - `AddTransient<IServiceA, ClassServiceA>()`
- Poprzez `AddSingleton<IServiceA, ClassServiceA>()`: przy pierwszej potrzebie tworzony jest jeden obiekt typu `ClassServiceA` i każdy inny obiekt z wstrzykniętym w konstruktorze interfejsie `IServiceA` otrzyma ten jeden, jedyny obiekt. Obiekt zostanie usunięty dopiero przy zamykaniu aplikacji
- Poprzez `AddScoped<IServiceA, ClassServiceA>()`: w ramach jednego żądania stworzona zostanie jedna instancja serwisu klasy `ClassServiceA`. Każde kolejne żądanie spowoduje wytworzenie nowej instancji tej klasy.
- Poprzez `AddTransient<IServiceA, ClassServiceA>()`: każda potrzeba skorzystania z serwisu (np. tworzenie klasy ze wstrzykniętym tym serwisem) powoduje wygenerowanie nowej instancja serwisu klasy `ClassServiceA`.
- Różnica między dwoma ostatnimi sposobami jest widoczna, gdy serwisy w kontenerze są między sobą powiązane .
- Załóżmy, że do kontenera serwisów dodano również serwis:
 - `AddTransient<IServiceB, ClassServiceB>()`
- **Dodatkowo 1:** w konstruktorze `ClassServiceB` wstrzyknięty jest `IServiceA`.
- **Dodatkowo 2:** W kontrolerze `XController` wstrzyknięto w konstruktorze `IServiceA` i `IServiceB`.
 - Jeśli `IServiceA` został dodany poprzez `AddScoped`, to dla `XController` będzie stworzona jedna instancja klasy `ClassServiceA`, natomiast przy tworzeniu `ClassServiceB` będzie wstrzyknięta **ta sama** instancja klasy `ClassServiceA`. Po obsłudze żądania ta instancja „ginie”.
 - Jeśli `IServiceA` został dodany poprzez `AddTransient`, to dla `XController` będzie stworzona jedna instancja klasy `ClassServiceA`, natomiast przy tworzeniu `ClassServiceB` dla tego kontrolera, będzie stworzona **kolejna** instancja `ClassServiceA`. Po obsłudze żądania obydwie instancje „giną”.

Działanie dla addScoped()

```
Container.AddScoped<IServiceA, ClassServiceA>()  
Container.AddTransient<IServiceB, ClassServiceB>()
```

```
public ClassServiceA(){ ... } // constructor
```

```
public ClassServiceB(IServiceA servA){ ... } // constructor
```

```
public Xcontroller(IServiceA servA, IServiceB servB)
```

Creations in Container (pseudocode)

```
var A1=new ClassServiceA();  
var B1=new ClassServiceB(A1);  
var controller=new Xcontroller(A1,B1);  
// then execute an action for XController
```

Działanie dla addTransient()

```
Container.AddTransient<IServiceA, ClassServiceA>()
```

```
Container.AddTransient<IServiceB, ClassServiceB>()
```

```
public ClassServiceA(){ ... } // constructor
```

```
public ClassServiceB(IServiceA servA){ ... } // constructor
```

```
public Xcontroller(IServiceA servA, IServiceB servB)
```

Creations in Container (pseudocode)

```
var A1=new ClassServiceA();  
var B1=new ClassServiceB(A1);  
var A2=new ClassServiceA();  
var controller=new Xcontroller(A2,B1);  
// then execute an action for XController
```