Data Access Layer & Razor Pages

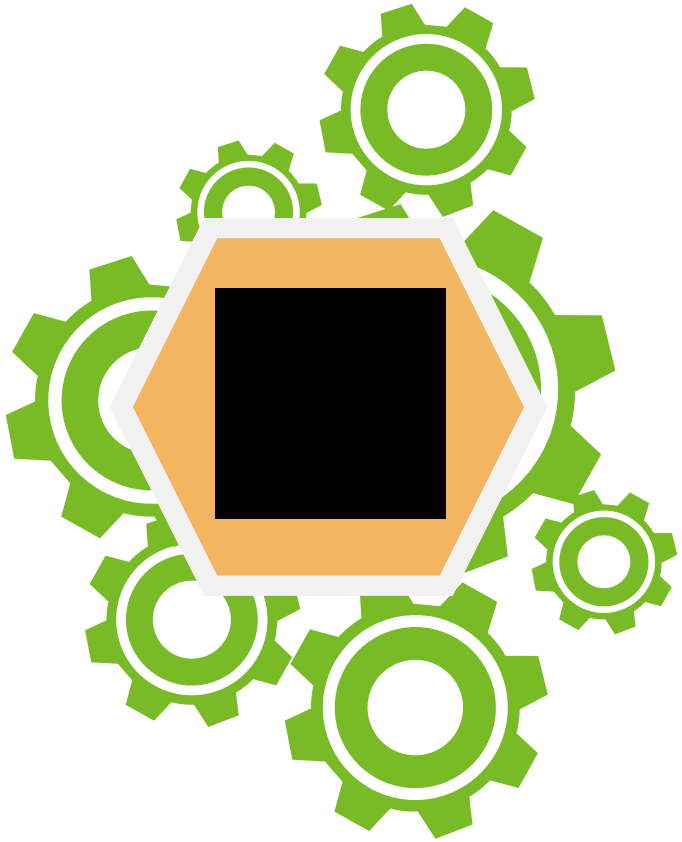
PV179

Obsah

Data Access Layer

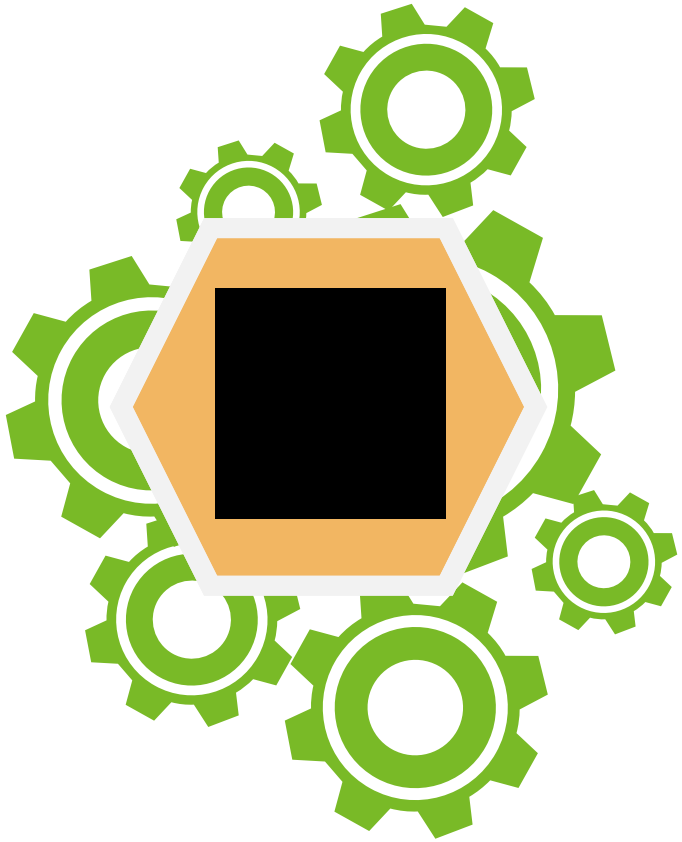
"Teoria", ale zrozumitelne

Data Access Layer



- Ukladanie dát
- Získavanie dát
- Práca s Databázou

Data Access Layer



Pripomenutie z prednášky

- Ako developeri, potrebujeme pracovať s dátami a pri práci s dátami potrebujeme pracovať s nejakým uložiskom. Najčastejšie sa jedna o databázu.
- Pri práci s databázou musíme dávat pozor na to, čo databázi posielame, ako dané dáta ukladáme a čo všetko vôbec uživatel môže vidieť.
- Každí zvás, by už mal mať za sebou PB154 (základy DB systémov), kde ste si vyskúšali napísať nejaké SQL Querries.
- Cielom DAL je zjednotiť prácu nad databázou a to tak, ze všetky operacie nad databázou povedu cez túto vrstvu.
- DAL nieje len abstrakcia and DB, ale nad akýmkolvek úložiskom (v tomto predmete budeme používať DB)

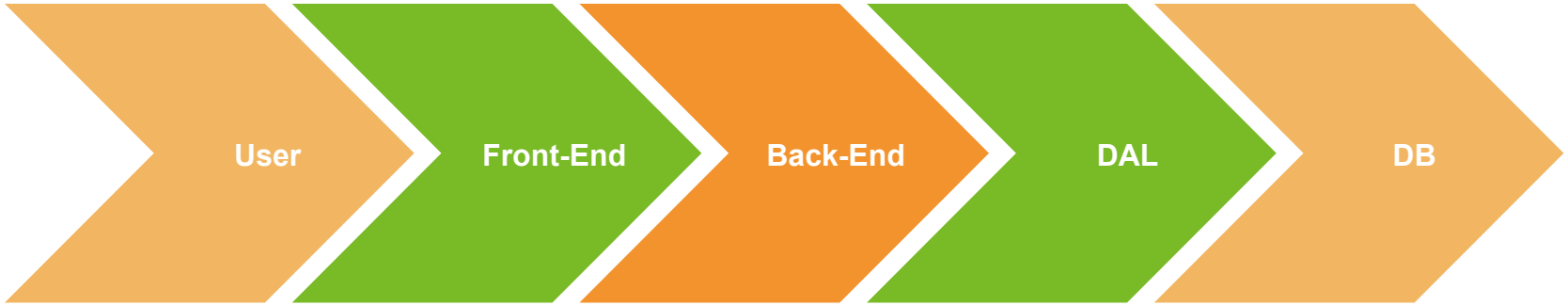
3 Vrstvova Architektura

Front-End

Front-End poskytuje formulár, pre uživatela, aby sa mohol zaregistrovat a zasiela data od uživatela na spracovanie do Back-Endu

Data Access Layer

Obdží dáta, stým, že ich ma uložiť do DB. Data zvaliduje, pričom napríklad overuje integritu dat.



Uživatel

Uživatel ma nejaké dáta, ktoré nám chce poskytnúť, napríklad meno a heslo

Back-End (Business Layer)

Spracuje data z Front-Endu. Napríklad overí, či uživatel poslal validne dáta, ktoré sú použitelné pre DAL, zahashuje heslo a napasuje ich na model, ktorý napríklad akceptuje DAL
(sneak peak do budúcna)

Databaza

Uloži dané dáta

Entity Framework

Entity Framework (Core)

Entity Framework (ďalej ako EF), je Object-Database mapper pre .NET.

Načo je EF dobrý?
- podporuje LINQ queries
- trackovanie zmien nad objektami
- Update db na základe trackovaných zmien
- zmeny schematu db cez migrácie.

EF Core zároveň funguje pre veľmi veľa RELAČNÝCH databáz, niektore znich su napriklad:
- SQLite
- MySQL
- PostgreSQL
- Azure Cosmos DB

Entity Framework

Uzitocne EF Nuget Balicky

- Microsoft.EntityFrameworkCore
  - Základný balíček, pre fungovanie s EF
- Microsoft.EntityFrameworkCore.Proxies
  - Primárne používaný pre LazyLoading
- Microsoft.EntityFrameworkCore.SqlServer
  - Dovoluje EF aby bol používaný spolu s Microsoft SQL Serverom
- Microsoft.EntityFrameworkCore.Tools
  - Primárne používaný pre managovanie migrácii, generovanie kódu pre DBContext a entity pre DB a reverse-engineering schémat db.

Data Access Layer

Model

- reprezentuje jednu tabulku (inštancia Entity) v DB.
- Po vytvorení objektu, ktorý je modelom, ho môžeme poslať do DB a tieto data sa nám uložia v DB a s objektom môžeme ďalej pracovať.

DBContext

- Jedna sa o abstrakciu jedneho konkretneho pripojenia do DB s informaciami o obsahu DB (DBSets – tabulky)
- Instancia resprezentujuca kombinaciu Unit Of Work a Reposotiry Patternu (na buducej hodine), ktora sluzi na vytvaranie queries, ktore nasledne pojdu do DB.
- Priklad: Vytvorime 2 uzivatelov a nasledne ich chceme poslat do DB. Staci ich pridat do DBSetu a ulozit zmeny.

DBSet

- Reprezentuje kolekciu entit ROVNAKEHO TYPU v danom DBContexte, ktore mozeme “queriunut” do DB (Jednoducho povedane, jedna sa o jednu tabulku v DB)
- Priklad: DbSet<User> reprezentuje tabulku v database, ktoru my vidime ako kolekciu modelov typu User

Príklad - Model

```
namespace PizzaShopDAL.Models
{
    public class Pizza : ScalableEntity
    {
        [MaxLength(64)]
        public string Name { get; set; }

        [Column(TypeName = "decimal(6, 2)")]
        public decimal Price { get; set; }

        public virtual List<PizzaIngredience> PizzaIngrediences { get; set; }
    }
}
```

Tvorba Query

```
using (var context = new PizzaShopDBContext())
{
    if (!context.Pizza.Where(a => a.Name == Pizza.Name).Any())
    {
        context.Pizza.Add(new Pizza
        {
            Name = Pizza.Name,
            Price = 19.99M
        });
        context.SaveChanges();
    }
}
```

Connection String sa v tomto príklade berie implicitne z Configu (najcastejsie sa jedna o appsettins.json)

Praca s Databazou

- context.Database.EnsureCreated() -> v prípade, že databáza nieje pre daný kontext vytvorená, tak je vytvorená s celou jej schémou a zároveň je zvalidovaná. POZOR, EnsureCreated nevyužíva migracie a zároveň snimi nemože byť upravovaná.

- DbContext.Database.Migrate() -> v prípade, že databáza neexistuje, vytvorí ju a aplikuje naňu všetky migrácie. V prípade, že už databáza existuje, ale chýbajú jej nejaké posledné migrácie, tak ich vykoná.

- Využitie EnsureCreated:
  - Testovacia DB
  - Dáta ktoré sa možu často menit (ne-relačné DB)

- Dôležité príkazy (píšeme do Package Manager Console):
  - Add-Migration <Meno Migracie>
  - Update-Database

- Alternatívy:
  - dotnet ef migrations add <Meno Migracie>
  - dotnet ef database update

Separacia Migracii

- Môže sa nám stať, že budeme chcieť oddeliť Migrácie a Modely od nášho DAL. Vtomto prípade, nám nebudú stačiť informácie z predchádzajúceho slidu.

- V našom novom projekte je zapotreby naconfigurovat assembly nasledovne:

4. Configure the migrations assembly:

C# Kopiraj

```
services.AddDbContext<ApplicationDbContext>(
    options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection"),
            x => x.MigrationsAssembly("WebApplication1.Migrations")));
```

- Zároveň budeme potrebovať nový parameter do nášho Add-Migration commandu

```
Add-Migration NewMigration -Project WebApplication1.Migrations
```

Task 01

Poďme si vyskúšať, čo sme práve videli

Task 01 - Zaujímavosti

Poďme si zopakovať, s čím sme práve pracovali

One-To-Many

- V úlohe nebolo potrebné konfigurovať One-To-Many relationship z dôvodu, že ak dodržiavame konvencie EF Core, tak to EF zvládne automaticky sám. V prípade, že by sme ich nedodržiavali, museli by sme použiť Fluent API s Has/With patternom.

- Príklad Has/With patternu:

```
modelBuilder.Entity<OrderItem>()
        .HasOne(pi => pi.Order)
        .WithMany(p => p.Items)
        .HasForeignKey(a => a.OrderId);
```

Seeding DB

- Seedovanie DB slúži ktomu, aby sme mali istotu, že ak spustim aplikaciu s čistou / starou DB, tak budeme mať k dispozícií základné data v oboch týchto databázach

- V našom projekte sa nám seeduje databáza v metóde `OnModelCreating`. To znamemá, že v momente, kedy sa využíva fluent API (presnejšie ModelBuilder API) na konfiguráciu modelov.

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new OrderConfiguration());
}

1 reference
public class OrderConfiguration : IEntityTypeConfiguration<Order>
{
    0 references
    public void Configure(EntityTypeBuilder<Order> builder)
    {
        builder.HasKey(o => o.Id);
        builder.Property(t => t.Person)
               .IsRequired();
        // seeding
    }
}
```

- Doporučuje sa konfiguráciu (a teda aj seedovanie DB) rozdelovať do osobitných súborov.

- Pri akejkolvek zmene v seedovacich dátach, je potrebné vytvoriť novú migráciu a updatnut databázu s danou migráciou, aby sa dané dáta zobrazili v DB.

One-To-Many (Seeding)

```
modelBuilder.Entity<Pizza>().HasData
(
    new Pizza {
        Id = 1,
        Name = "Hammito",
        ShapeId = 1,
        Height = 10,
        Width = 10,
        Price = 249.99M,
    }
);

modelBuilder.Entity<Order>()
    .HasData(new Order { Id = 1, PersonId = 1 });

modelBuilder.Entity<OrderItem>()
    .HasData(new OrderItem { Id = 1, Amount = 2, PizzaId = 1, OrderId = 1 });
}
```

Konzolová Appka

- Po vytvorení konzolovej applikácie bolo zapotreby, aby ste si pridali project reference
- Následne bolo zapotreby, aby ste získali dáta použitím DBContextu.
- Je dôlezite, aby ste vedeli, že po zániku DBContextu už nieje možné pracovať s databázov a teda musíte vašu query executnuť a získať dáta vo formáte “IEnumerable<…>”

```
using PizzaShopDAL.Data;
using PizzaShopDAL.Models;

IQueryable<Order>? orders;

using (var context = new PizzaShopDBContext())
{
    orders = context.Order.Where(a => a.Id < 10);
    // Will work as intended
    Console.WriteLine($"Order -> {orders.Count()}");
}

// Will result in error (DBContext is Disposed)
Console.WriteLine($"Order -> {orders.Count()}");
```

```
using PizzaShopDAL.Data;
using PizzaShopDAL.Models;

IEnumerable<Order>? orders;

using (var context = new PizzaShopDBContext())
{
    orders = context.Order.Where(a => a.Id < 10).ToList();
    // Will work as intended
    Console.WriteLine($"Order -> {orders.Count()}");
}

// Will be fine
// CAREFUL!, .AsEnumerable() DOES NOT EXECUTE THE QUERY
Console.WriteLine($"Order -> {orders.Count()}");
```

Razor Pages

"Teoria", ale zrozumitelne

Razor Pages

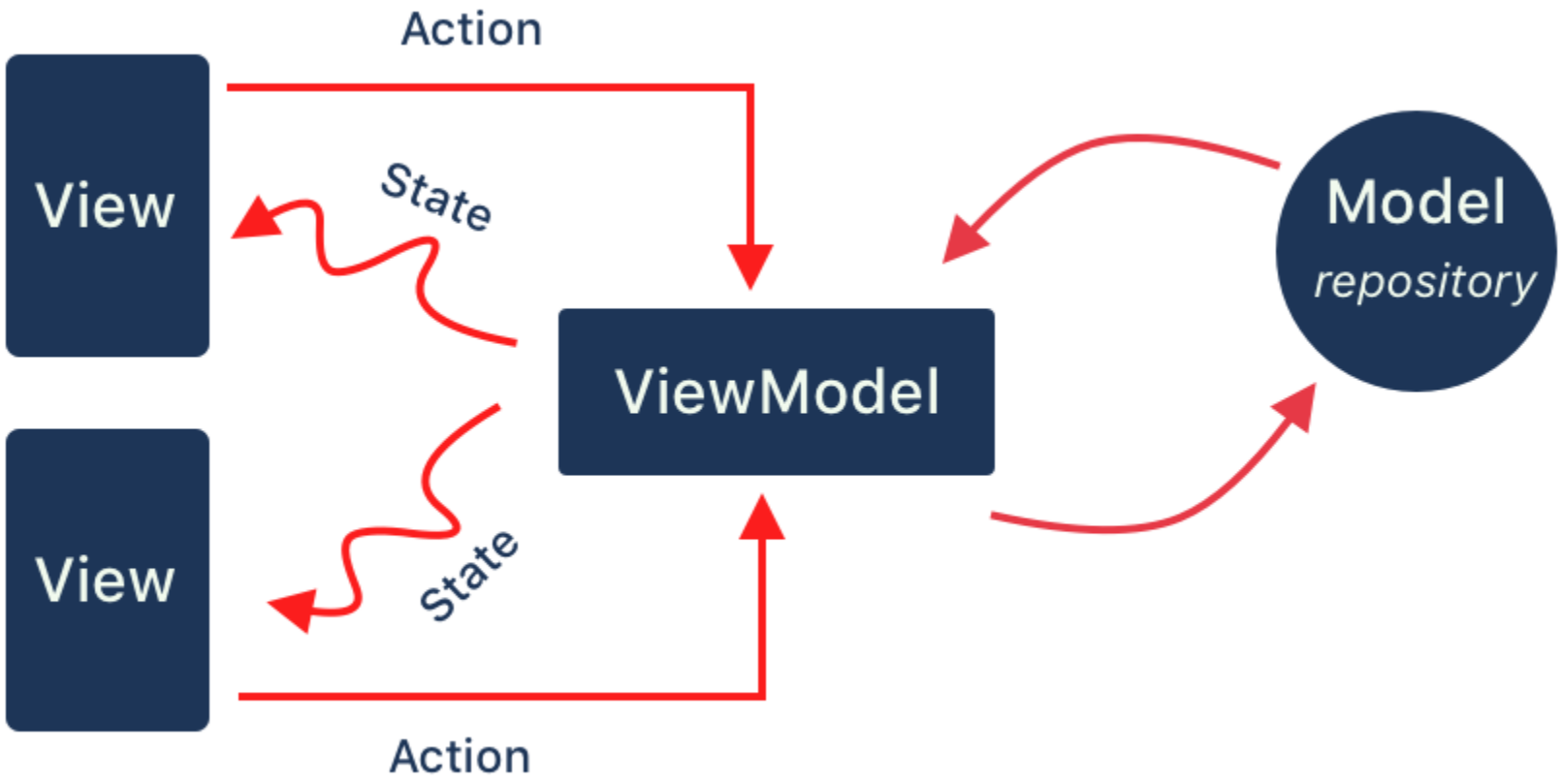
- page-based model, kde UI a business logika sú separované, avšak stále vrámci jednej stránky (page)

- Čo to vlastne znamená?
  - Využívame MVVM pattern, vďaka ktorému vieme separovať business logiku od UI, avšak stale to držat vrámci jednej “page”.

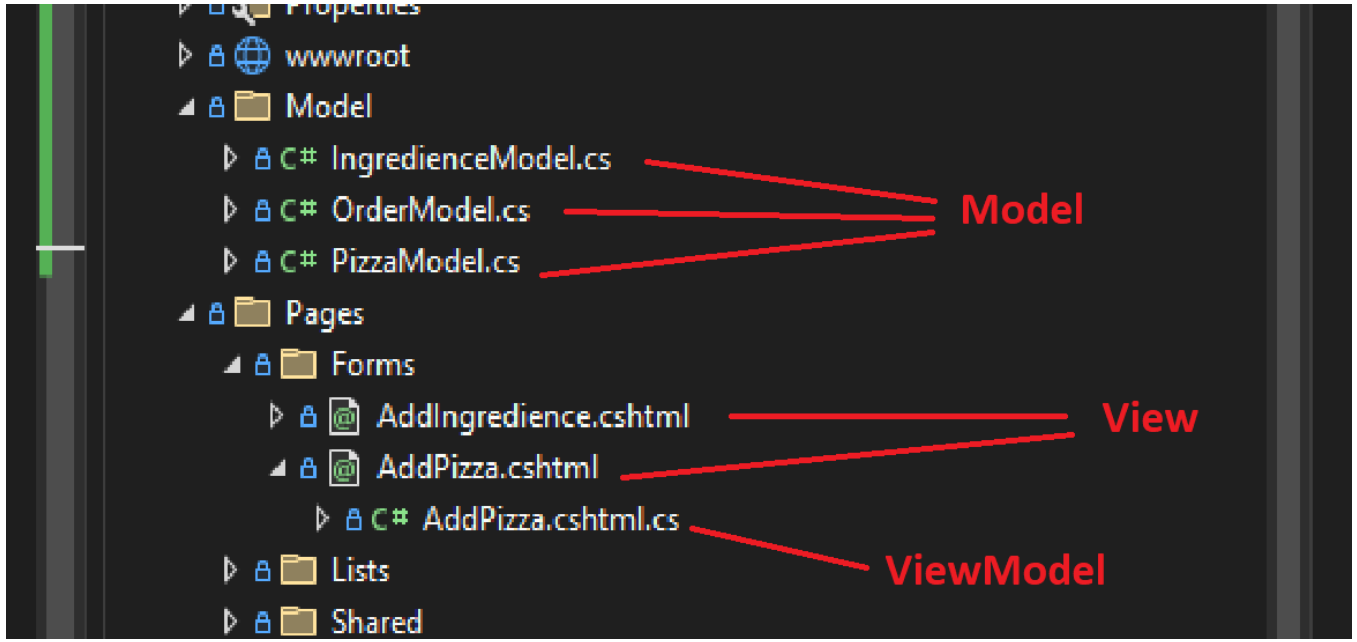
- Motivácia na ďalšie cvicenie:
  - Často sa stáva, že potrebujeme rovnaký kód na viacerých pages, len s malými úpravami. Toto sa rieši tak, že sa vytvorí ďalšia vrstva, tiež nazývaná ako “business layer”, kde sa nachádzajú napríklad services, ktoré využívame na získavanie dát a ich modifikáciu

https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio

MVVM Architektúra



MVVM Architektura



- Model: Rovanko ako v DAL slúži ako reprezentácia dát. V tomto prípade to nemusí byt čisto DB entita, ale môže to byť napríklad malý kúsok danej entity (Fieldy, ktoré v daný moment výhradne potrebujem)

- View: Front-End, v tejto časti sa generuje HTML, ktoré daný uživatel uvidí na Front-Ende. Je možné tvoriť aj “partial views”, ktoré sú akoby podčasť stránky, ktorú si môžeme na zavolanie získat jedným razor tagom ;)

- ViewModel: Logika, ktorá spája Model a View. Presnejšie sa jedná o reprezentáciu dát a ich správania, spojené s jedným špecifickým View.

Razor Pages - M

```
namespace PizzaShopFE.Model
{
    1 reference
    public class IngredienceModel
    {
        3 references
        public string Name { get; set; }
    }
}
```

- Vždy je lepšie vytvárať vlastne modely na Front-End a nepoužívať tie databázove z dôvodu, že ak nastane nejaký problém, napríklad sa nám zmení ID alebo nám v medzičase niekto zmaže našu entitu v DB, tak začnú vznikať konflikty.

- Môžeme si všimnúť, že PizzaShopFE model (Razor Page Model) neobsahuje PizzaIngrediences, ktoré obsahuje PizzaShopDAL model

```
namespace PizzaShopDAL.Models
{
    16 references
    public class Ingredience : BaseEntity
    {
        [MaxLength(64)]
        9 references
        public string Name { get; set; }

        1 reference
        public virtual List<PizzaToIngredience> PizzaIngrediences { get; set; }
    }
}
```

Razor Pages - V

```
@page
@model PizzaShopFE.Pages.Forms.AddIngredienceModel
@{
    var name = Model?.Ingredience?.Name ?? null;
}

@if (name == "Ham")
{
    <h1>Its the O'Mighty Ham</h1>
}
else if (name != null)
{
    @Html.DisplayFor(a => a.Ingredience.Name)
}

<form method="post">
    <p>Ingredience Name : </p>
    <input type="text" asp-for="Ingredience.Name" placeholder="Fish"/>

    <button type="submit">Send!</button>
</form>
```

- @page - requesty sú riešené napriamo (nevolá sa nejaký ďalší controller)
- @model – definuje model, ktorý bude použitý pre dané View (aby sme mohli pracovať s dátami)
- @{} – code block, do ktorého možeme písať C# kód
- @Something {} – Za something vieme nahradiť C# code block akcie ako napríklad if, for alebo foreach.
- Asp-for=“-” – Bind property na formulárový input
- Method=“POST” – formular odoslať cez POST (teda nie ako parameter)

Razor Pages - V

- Shared/_Layout.cshtml – Template pre každú webstránku. Jedná sa o kostru webstránky, kde sa definuje telo stránky za použitia @RenderBody(), kde sa neskôr bude nachádzať kód z nášho View.

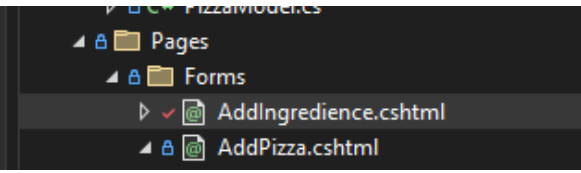
- Error.cshtml – Zobraznie webstránky v momente, keď nastane nejaký server-side error

- `ViewData["Title"] = "Home page";` Nastaví Webpage Title na “Home Page”

```
<meta name="viewport" content="width=device-width,
<title>@ViewData["Title"] - PizzaShopMVC</title>
<link rel="stylesheet" href="~/lib/bootstrap/dist/
```

- Asp-page=“--” po nastavení cesty z korena (Page Directory) vykoná redirect na cestu v page

```
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-page="/Forms/AddIngredience">Add Ingredience</a>
</li>
```



- @Html.<TextBox/CheckBox/…>For() – HtmlHelper, ktorý obsahuje rozšírujúce metódy na generovanie HTML kódu pre razor view. Príklad: @Html.TextBoxFor(m => m.StudentName)

```
<input id="StudentName" name="StudentName" type="text" value="" />
```

Razor Pages - VM

```
5 references
public class AddPizzaModel : PageModel
{
    [BindProperty]
    4 references
    public PizzaModel PizzaOrder { get; set; }

    0 references
    public void OnGet()
    {
    }

    0 references
    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
            return Page();
        return RedirectToPage("../Index");
    }
}
```

- [BindProperties] / [BindProperty] Attribúty slúžia na “bind” z ViewModelu do View. Teda je možné tieto properties accessnuť priamo vo View
- OnGet -> Vykoná nejakú akciu na základe GET Requestu
- OnPost -> Vykoná nejakú akciu na základe POST Requestu
- ModelState.IsVald -> zvaliduje model pomocou validačných attribútov a tým určí, či dané dáta sú správneho typu (napríklad hodí false v prípade, že v property typu int je string)

Razor Pages - VM

```
5 references
public class AddPizzaModel : PageModel
{
    [BindProperty]
    4 references
    public PizzaModel PizzaOrder { get; set; }

    0 references
    public void OnGet()
    {
    }

    0 references
    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
            return Page();
        return RedirectToPage("../Index");
    }
}
```

- Model binding: process pri ktorom sa vezmú hodnoty z HTTP requestu a namapujú sa na “handler method” parameter alebo “PageModel” property.
- “handler method” – OnGet(), OnPost() …
- “PageModel” – View Model
- Model binding system:
  - Získava dáta zo zdrojov, ako napríklad route data, prvkov formulárov a query stringu
  - Poskytuje data Controllerom (neskôr v predmete) a Razor Pages ako parametre metód alebo verejné properties
  - Konvertuje string dáta do .NET typov
  - Updatuje properties komplexných typov

https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-6.0

Task 02

Poďme si vyskúšať, čo sme práve videli

Task 02 - Zaujímavosti

Poďme si zopakovať, s čím sme práve pracovali

LazyLoading

```
3 references
public Dictionary<Pizza, List<Ingredience>> Pizzas { get; set; }
0 references
public void OnGet()
{
    Pizzas = new();
    using (var context = new PizzaShopDBContext())
    {
        foreach (var pizza in context.Pizza)
        {
            var ingrediences = new List<Ingredience>();

            foreach (var ingredience in pizza.PizzaIngrediences)
            {
                ingrediences.Add(ingredience.Ingredience);
            }

            Pizzas.Add(pizza, ingrediences);
        }
    }
}
```

- LazyLoading je technika, ktorá umožňuje načítanie dát až v momente, keď sú dané dáta potrebné.
- Jedná sa teda naozaj o “lenivé” načítanie dát.
- Častým problémom je, že sa snažíme pristúpiť k nejakým dátam v momente, kedy uz nemáme DB Connection (DBContext je Disposed)
- Zároveň to môže spôsobovať spomalovanie aplikácií v momentoch, kedy by sme to nemuseli čakať (myslíme si, že máme nejakú entitu k dispózicií, avšak ona sa ešteln tahá z DB)

Ďakujem za pozornosť

A dúfam, že ste sa toho veľa naučili ☺

Free PPT Templates

Zaujimave linky, ktore dopoducujem navstivit:

- https://www.learnrazorpages.com/razor-pages/pagemodel#:~:text=View%20Models&text=It%20is%20a%20self%2Dcontained,is%20also%20the%20view%20model.

- https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio

- https://www.learnentityframeworkcore.com/configuration/fluent-api