Popis labů ke kurzu Architektura webových aplikací

Osnova

- Struktura webových MVC aplikací – jednotlivé vrstvy
- Datová vrstva a zajímavé vzory
    - Entity Framework, normální formy
    - Repositories
- Business vrstva a zajímavé vzory
- Prezentační vrstva
- Dependency injection pomocí Windsor Castle
- Testování aplikací a jak aplikace psát testovatelně

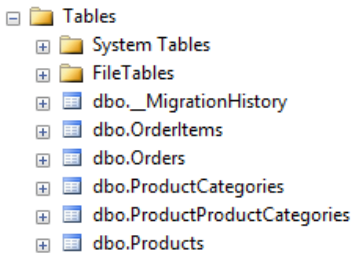
1. Založení projektu a základních vrstev

1. Naše aplikace bude jednoduchou verzí e-shopu, bude třeba tedy uchovávat produkty, kategorie a objednávky (velmi zjednodušeně). Produkt by měl mít Id, Jméno, Popis, Kategorie a cenu. Kategorie produktu by měla mít Id, Jméno a seznam produktů, které v ní jsou. Mezi Produktem a Kategorií by měla být M:N vazba. Není třeba uvažovat hierarchii kategorií. Objednávka pak obsahuje Id, doručovací adresu, celkovou cenu a seznam produkt (třídu OrderItem, který má vazbu na produkt a množství).
2. Z L:\MvcDemoShop01\netcore\ si zkopírujte k sobě solution a projekty, které otevřete ve Visual Studiu.
3. Projekt obsahuje model v projektu Model a základní MVC views a controllery (CatalogController) v projektu Web:
    a. **Index** sloužící pro zobrazování seznamu kategorií.
    b. **Category(int id)** sloužící k zobrazení produktů dané kategorie. Dle předaného ID by měla vytvořit nějaká testovací data typu ViewModels.CategoryDetailViewModel.

2. Entity Framework jako ORM

1. Nainstalujte nuget Microsoft.**EntityFrameworkCore** a **Microsoft.EntityFrameworkCore.SqlServer** do projektu DataAccessLayer (buď přes GUI nebo pomocí příkazu Install-Package Microsoft.EntityFrameworkCore.SqlServer –Project DataAccessLayer v package manager konzoli)
2. Nainstalujte nuget **Microsoft.EntityFrameworkCore** a **Microsoft.EntityFrameworkCore.SqlServer** do projektu Web (buď přes GUI nebo pomocí příkazu Install-Package Microsoft.EntityFrameworkCore.SqlServer –Project Web v package manager konzoli)

3. Vytvořte DB Context jménem ShopDbContext (namespace DataAccessLayer). Tento DB Context je odvozen od Systém.Data.Entity.DbContext a obsahuje pro každou entitu (tabulku property ve tvaru `public DbSet<Order> Orders { get; set; }`
4. Této třídě je třeba přidat bezparametrický konstruktor volající base constructor pro určení názvu connection stringu:
```
public ShopDbContext(DbContextOptions options) : base(options) { }
```
5. Je třeba nakonfigurovat naši vazební tabulku pomocí override metody OnModelCreating:
```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<ProductCategoryProduct>()
        .HasKey(x => new {x.ProductCategoryId, x.ProductId});
}
```
6. Do appsettings.json projektu web je třeba přidat connection string:
```
  "ConnectionStrings": {
   "Default": "Server=(localdb)\\mssqllocaldb; Database=MvcDemoShopCore;
Integrated Security=true;"
  }
```
7. Nastavte EF v souboru **Startup.cs** projektu **Web** přidáním řádky do **ConfigureServices:**
```
services.AddDbContext<ShopDbContext>(x =>
x.UseSqlServer(Configuration.GetConnectionString("Default")));
```
8. Je třeba vytvořit naši první migraci příkazem add-migration init
9. Ověřte, že vypadá OK.
10. Spusťe ručně update databáze pomocí příkazu v package manager consoli: `update-database -ProjectName DataAccessLayer -StartUpProjectName Web`
11. Spusťe SQL Server Management Studio a připojte se na naši LocalDB instanci – tedy na server (LocalDb)\MSSQLLocalDB. Zde by měla existovat databáze MvcDemoShopCore s našimi tabulkami:



12. Vyplňte tabulky Product, ProductCategoryProduct a Products nějakými testovacími daty.
13. Upravte CatalogController projektu Web tak, aby neměl data zadrátovaná v kódu, ale četl je z databáze:
    a. V konstruktoru si necháme injectnout ShopDbContext:
    ```
    public CatalogController(ShopDbContext dc) { this.dc = dc; }
    ```
    b. Akce index bude číst data z tabulky ProductCategories:

```
public ActionResult Index()
{
    TopCatalogViewModel data = new TopCatalogViewModel()
    {
        Categories = dc.ProductCategories.Select(c => new
CategoryListViewModel
            {
                Id = c.Id,
                Name = c.Name
            })
            .ToList()
    };
    return View(data);
```

c. Akce Category pak data z tabulky Product dle zvolené kategorie.

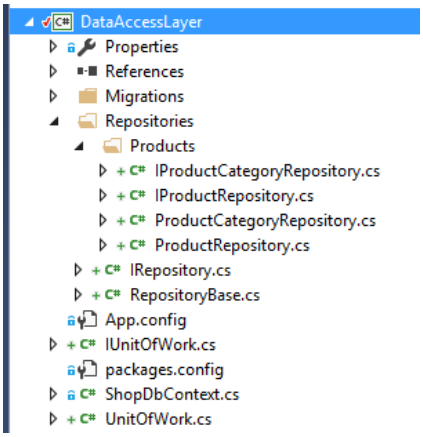
14. Ozkoušejte, že vše funguje. Ukázkové řešení je dostupné na L:\MvcDemoShop02\netcore

3. Vytvoření Repository a Unit of Work

1. Vytvořte bázová rozhraní IRepository<TEntity> (pro klíče typu int) a IUnitOfWork odpovídající

```
public interface IRepository<TEntity>
        where TEntity : class
{
        TEntity GetById(int id);
        List<TEntity> GetAll();
        void Create(TEntity entity);
        void Update(TEntity entity);
        void Delete(TEntity entity);
}
public interface IUnitOfWork : IDisposable
{
        void SaveChanges();
}
```

2. Vytvořte RepositoryBase s generickou implementací pro EntityFramework. DbSet získáte z DbContextu metodou Set<>(). Vytvořte UnitOfWork implementující IUnitOfWork za pomoci SaveChanges metody Entity Frameworku. Repository bude v konstruktoru přijímat DB Context (ShopDbContext).
3. Implementujte repository pro ProductCategory (ProductCategoryRepository) a pro Product (ProductRepository). ProductRepository by měl mít ještě metodu pro získání produktů dle ID kategorie. Vrstva DataLayer by potom měla vypadat přibližně takto:



4. Ukázkové implementace jsou k dispozici na L:\MvcDemoShop03\netcore
5. Upravte CatalogController tak, aby používal nově vytvořené repository.

4. Vytvoření byznys služeb

Cílem cvičení je přesunout logiku z controllerů do byznys služeb.

1. Vytvořte ProductCategoryService v projektu BusinessLayer.
2. Pro jednoduchost bude tato služba dostávat v konstruktoru naší ShopDbContent a pomocí něj vytvářet potřebné repository, které uloží do fieldu třídy:

```
public ProductCategoryService(ShopDbContext dbContext)
{
        productRepository = new ProductRepository(dbContext);
        productCategoryRepository = new ProductCategoryRepository(dbContext);
}
```

3. Implementujte metodu GetRootCatalog, který vrátí TopCatalogViewModel, který bude naplněn pomocí repository – jde v podstatě o přesunutí logiky z akce CatalogController.Index do této metody.
4. Upravte akci CatalogController.Index tak, aby používala nově vytvořenou službu.
5. Implementujte metodu GetProductsByCategory(int categoryId), který vrátí CategoryDetailViewModel, který bude naplněn pomocí repository – jde v podstatě o přesunutí logiky z akce CatalogController.Category(int id) do této metody.
6. Upravte akci CatalogController.Category tak, aby používala nově vytvořenou službu.
7. Založte nové UI modely Cart a CartItem do projektu ViewModels:

```
[Serializable]
public class Cart
{
	public List<CartItem> Items { get; set; } = new List<CartItem>();
}
[Serializable]
public class CartItem
{
	public int ProductId { get; set; }
	public int Quantity { get; set; }
}
```

8. Vytvořte službu OrderService v BusinessLayer projektu.
9. Pro jednoduchost bude tato služba dostávat v konstruktoru naší ShopDbContent a pomocí něj vytvářet potřebné repository, které uloží do fieldu třídy (jako v předchozím případě).
10. Implementujte metodu `public Cart AddToCart(Cart cart, int productId)`, která vezme existující košík (nebo může null, záleží na implementaci) a přidá do něj potřebný produkt (případně zvýší množství produktu v košíku).
11. Implementujte metodu `public void CreateOrder(Cart cart, string address)`, který vezme košík a vytvoří z něj objednávku – tedy založí entity Order a OrderItem a uloží je do databáze (pomocí OrderRepository a UnitOfWork).
12. Na některé cvičné testovací akci zkuste, jestli nově vytvořené metody na OrderService fungují OK.
13. Cvičně ukázkové řešení je na L:\MvcDemoShop04\netcore

5. Vytvoření cache aplikační služby a vzor dekorátor

1. Vytvořte rozhraní ICacheService v projektu BusinessLayer pro naši cachovací službu:

```
	public interface ICacheService
{
	object Get(string key);
	void Add(string key, object value, DateTime expiration);
	/// <summary>
	/// Vrátí hodnotu z cache, pokud existuje, nebo spusti <paramref
name="valueFactory"/>, hodnotu uloží do cache a vrátí ji
	/// </summary>
	T GetOrAdd<T>(string key, DateTime expiration, Func<T> valueFactory)
		where T : class;
}
```

2. Implementujte toto rozhraní v nové službě CacheService. Je nejlepší pro ukládání použít již vestavěnou MemoryCache. Tu získáte z DI kontejneru tím, že přidáte konstruktor:

```
public CacheService(IMemoryCache memoryCache) { this.memoryCache = memoryCache;
}
```

3. Implementujte metody Get a Add této služby za použití injectnutí IMemoryCache (využijte extension metody Get a Set).
4. Nastavte MemoryCache v souboru **Startup.cs** projektu **Web** přidáním řádky do **ConfigureServices:**
   `services.AddMemoryCache();`
5. Extrahujte rozhraní IProductCategoryService z již existující služby ProductCategoryService (refactoring Extract Interface vytvoří takový interface, který bude mít všechny public metody třídy, z níž je extrahováno), vznikne tedy rozhraní s metodami `TopCatalogViewModel GetRootCatalog(); a CategoryDetailViewModel GetProductsByCategory(int categoryId);.`
6. Vytvořte dekorátor pro ProductCategoryService, který se bude jmenovat např. CachedProductCategoryService, a který přijímá instanci rozhraní IProductCategoryService a „zabaluje“ jednotlivá volání do cache. Pro přístup do cache použijte naši CacheService (ta lze vytvořt např v constructoru a uložit si ji do instančního fieldu CachedProductCategoryService. Př:

```
        public TopCatalogViewModel GetRootCatalog()
        {
                return cache.GetOrAdd("GetRootCatalog", DateTime.Now.AddHours(1),
service.GetRootCatalog);
        }
```

7. Upravte CategoryController tak, aby jeho akce nepoužívaly služby ProductCategoryService přímo, nýbrž přes náš nově vytvořený dekorátor CachedProductCategoryService.
8. Ukázkové cvičné řešení je jako obvykle k dispozici na L:\MvcDemoShop05\netcore

6. Byznys logika pomocí actorů

1. Nareferencujte z projektů BusinessLayer a Web nuget **Akka** a **Akka.DI.Core**.

V projektu BusinessLayer:

2. Vytvořte složky **Actors/Implementations** a **Actors/Messages**
3. Vytvořte třídu **Actors/Messages/CatalogRequestMessage**, která bude sloužit jako zpráva pro Actora a bude prázdná.
4. Vytvořte třídu **Actors/Messages/CatalogResponseMessage**, která bude sloužit jako odpověd od actora a bude tedy obsahovat data ohledně katalogu, např.:

```
public class CatalogResponseMessage
{
        public List<string> Categories{ get; set; }
        public List<string> Products { get; set; }
}
```

5. Vytvořte nového aktora (třídu) **Actors/Implementations/CatalogActor**, který bude odvozen od Akka.Actor.ReceiveActor.
6. V konstruktoru vytvořte všechny potřebné repository, které si uložte do private fieldů aktora. Dále Zavolejte fci **Receive<CatalogRequestMessage>(HandleCatalogRequest),** čímž nastavíte metodu (zatím neexistující) jako handler pro naši CatalogRequestMessage.

```
public CatalogActor()
{
        var dbContext = new ShopDbContext();
        productRepository = new ProductRepository(dbContext);
        productCategoryRepository = new ProductCategoryRepository(dbContext);
        Receive<CatalogRequestMessage>(HandleCatalogRequest);
}
```

7. Naimplementujte **HandleCatalogRequest** tak, aby za pomoci repositories (vytvořených v konstuktoru) si vytáhla data z databáze, vytvořila CatalogResponseMessage obsahující vytažená data a tuto zprávu odeslala odesílateli Request zprávy (tj. Sender.Tell(result);). Celou logiku můžete obalit try-catch pro detekci chyb.

```
bool HandleCatalogRequest(CatalogRequestMessage message)
{
	try
	{
		var result = new CatalogResponseMessage()
		{
			Categories = productCategoryRepository.GetAll().Select(x =>
x.Name).ToList(),
			Products = productRepository.GetAll().Select(x =>
x.Name).ToList()
		};
			Sender.Tell(result);
	}
	catch (Exception e)
	{
		Sender.Tell(new Failure() { Exception = e});
	}
	return true;
}
```

8. Dále je potřeba vytvořit třídy, které budou deployovat actory. Nejdříve je potřeba vytvořit rozhraní **Actors/Implementations/IDeployer** :

```
public interface IDeployer
{
	void Deploy(ActorSystem actorSystem, IDependencyResolver
dependencyResolver);
}
```

9. Následně vytvořte deployera pro našeho CatalogActora:

```
public void Deploy(ActorSystem actorSystem, IDependencyResolver
dependencyResolver)
{
    var props = dependencyResolver.Create<CatalogActor>();
    actorSystem.ActorOf(props, ActorContracts.CatalogActorName);
}
```

10. Tím je logika hotova a je potřeba ještě dodělat infrastrukturu. Nejdříve vytvořte třídu **Actors/ActorContracts**, která bude obsahovat jména aktorů.

```
public static class ActorContracts
{
	public const string CatalogActorName = "catalogActor";
}
```

11. Nakopírujte soubor **L:\MvcDemoShop06\netcore\BusinessLayer\Actor\NetCoreDependencyResolver.cs** do projektu (BusinessLayer\Actor)
12. Následně vytvořte třídu **Actors/AkkaInfrastracture**, která bude spravovat ActorSystem a bude singleton (static), aby byla přístupná odevšad. Toto řešení potom vylepšíme pomocí Dependency Injection.

```
public class AkkaInfrastracture
{
	static readonly ActorSystem Actors;

	static AkkaInfrastracture()
```

```
        {
                Actors = ActorSystem.Create("DemoShop");
        }

        public static void InitAkka(IServiceCollection serviceCollection)
        {
                var deployers = Assembly.GetExecutingAssembly().GetTypes()
                        .Where(x => typeof(IDeployer).IsAssignableFrom(x) &&
x.IsClass && !x.IsAbstract)
                        .Select(t => (IDeployer)Activator.CreateInstance(t))
                        .ToList();
                 var propsResolver = new NetCoreDependencyResolver(serviceCollection,
Actors);

                 deployers.ForEach(x => x.Deploy(Actors, propsResolver));

        }

        public static Task<TResponse> Ask<TRequest, TResponse>(string actorName,
TRequest request)
        {
                var actor = Actors.ActorSelection("/user/" + actorName);
                return actor.Ask<TResponse>(request);
        }
}
```

Nyní je ještě potřeba zavolat novou infrastrukturu z MVC, následující kroky je tedy potřeba udělat v projektu Web:

1. Přidejte novou action metodu do **CatalogController**, která zavolá našeho nového actora a zpřístupního tak světu.

```
        public async Task<ActionResult> Api()
        {
                var result = await AkkaInfrastracture.Ask<CatalogRequestMessage,
CatalogResponseMessage>(ActorContracts.CatalogActorName, new CatalogRequestMessage());
                return Ok(result);
        }
```

2. Ve **Startup** v metodě **ConfigureServices** na konci zavolejte `AkkaInfrastracture.InitAkka(services);`
3. Zkuste zavolat /Catalog/Api

Zdroják je k dispozici na L:\MvcDemoShop06\netcore

7. Práce s asynchronními metodami v actoru

Nejdříve je potřeba si dodělat aspoň minimální podporu pro async metody v naší repository.

1. Do interface DataAccessLayer/Repositories/IRepository doplňte metodu

```
Task<List<TEntity>> GetAllAsync();
```

2. Do třídy DataAccessLayer/Repositories/RepositoryBase tuto metodu implementujte za pomoci Entitiy Frameworku.

```
public Task<List<TEntity>> GetAllAsync()
{
        return dbContext.Set<TEntity>().ToListAsync();
}
```

Teď je potřeba upravit CatalogActor tak, aby používal nově vytvořenou async metodu. Přestože lze v actorech používat await, tak by to dělat nebudeme, neboť během awaitu je actor blokován. Proto zvolíme strategii, kde spustíme jednolivé Tasky (z GetAllAsync) a jakmile tasky doběhnou, tak si jejich výsledek pomocí funkce PipeTo přesměrujeme jako zprávu na Self (aktuálně běžícího aktora).

3. Upravíme tedy metodu HandleCatalogRequest tak, aby výsledky přes PipeTo si posílala jako zprávu, abychom nemuseli čekat.

```
bool HandleCatalogRequest(CatalogRequestMessage message)
{
	try
	{
		var result = new CatalogResponseMessage();
		productCategoryRepository.GetAllAsync().ContinueWith(t =>
		{
			result.Categories = t.Result.Select(x => x.Name).ToList();
			return result;
		}).PipeTo(Self, Sender);
		productRepository.GetAllAsync().ContinueWith(t =>
		{
			result.Products = t.Result.Select(x => x.Name).ToList();
			return result;
		}).PipeTo(Self, Sender);
	}
	catch (Exception e)
	{
		Sender.Tell(new Failure() { Exception = e});
	}
	return true;
}
```

4. Dále je potřeba vytvořit funkci, která přijme ResponseMessage a zkontroluje, zda-li v ní jsou už všechna data.

```
bool HandleData(CatalogResponseMessage data)
{
	if (data.Categories != null && data.Products != null)
	{
		Sender.Tell(data);
	}
	return true;
}
```

5. Tento handler je pak třeba zaregistrovat v konstuktoru aktora:
`Receive<CatalogResponseMessage>(HandleData);`
6. Otestujte.

8. Mediatr

1. Do projektu BusinessLayer nainstalujte nuget **Castle.Windsor** a **MediatR**. Do projektu Web nainstalujte nuget **MediatR**. Do projektu ViewModels nainstalujde nuget **System.ComponentModel.Annotations.**
2. Vytvořte složky Mediators/Implementations a Mediators/Messages.
3. Vytvořte třídu pro generickou odpověď BusinessLayer/Mediators/Messages/**ActionResponseMessage**

```
public class ActionResponseMessage
{
	public bool Success { get; set; }
}
```

4. Vytvořte třídu pro zprávu BusinessLayer/Medidators/Messages/**NewProductMessage** s vlastnostmi pro nový produkt

```
public class NewProductMessage : IRequest<ActionResponseMessage>
{
	public string Name { get; set; }
	public string Description { get; set; }
	public List<int> Categories { get; set; }
	public decimal PriceWithoutVat { get; set; }
}
```

5. Vytvořte handler **BusinessLayer/MediatorsImplementations/NewProductHandler**, který bude umět zpracovávat zprávy typu NewProductMessage, bude tedy implementovat rozhraní IRequestHandler<NewProductMessage, ActionResponseMessage>.
6. V konstruktoru této třídy vytvořte všechny potřebné repository a uložte je do privátních fieldů třídy.

```
	public NewProductHandler()
	{
		var dbContext = new ShopDbContext();
		productRepository = new ProductRepository(dbContext);
		productCategoryRepository = new
ProductCategoryRepository(dbContext);
		uow = new UnitOfWork(dbContext);
	}
```

7. V metodě handle vytvořte instanci třídy Product (z DB modelu) a tu vložte do naší repository a uložte pomocí unit of work. Zpět je pak třeba vrátit instanci **ActionResponseMessage**.

```
	public Task<ActionResponseMessage> Handle(NewProductMessage request,
CancellationToken cancellationToken)
	{
		var newProduct = new Product
		{
			Description = request.Description,
			Name = request.Name,
			PriceWithoutVat = request.PriceWithoutVat,
			Categories = productCategoryRepository.GetAll().FindAll(x =>
request.Categories.Contains(x.Id))
		};
		productRepository.Create(newProduct);
		uow.SaveChanges();
		return Task.FromResult(new ActionResponseMessage() { Success = true });
	}
```

8. Zkopírujte a přidejte třídu **MediatorsInfrastructure** z L:\MvcDemoShop08\netcore\BusinessLayer/Mediators/MediatorsInfrastructure do projektu (na stjené místo jako je na L: disku).
9. V projektu ViewModels je potřeba udělat DTO **ProductDto** třídu pro naše API včetně data anotací pro validace

```
public class ProductDto
{
	[Required]
	public string Name { get; set; }
	[Required]
	public string Description { get; set; }
	public List<int> Categories { get; set; }
	[Range(1, 10000)]
	public decimal PriceWithoutVat { get; set; }
```

}

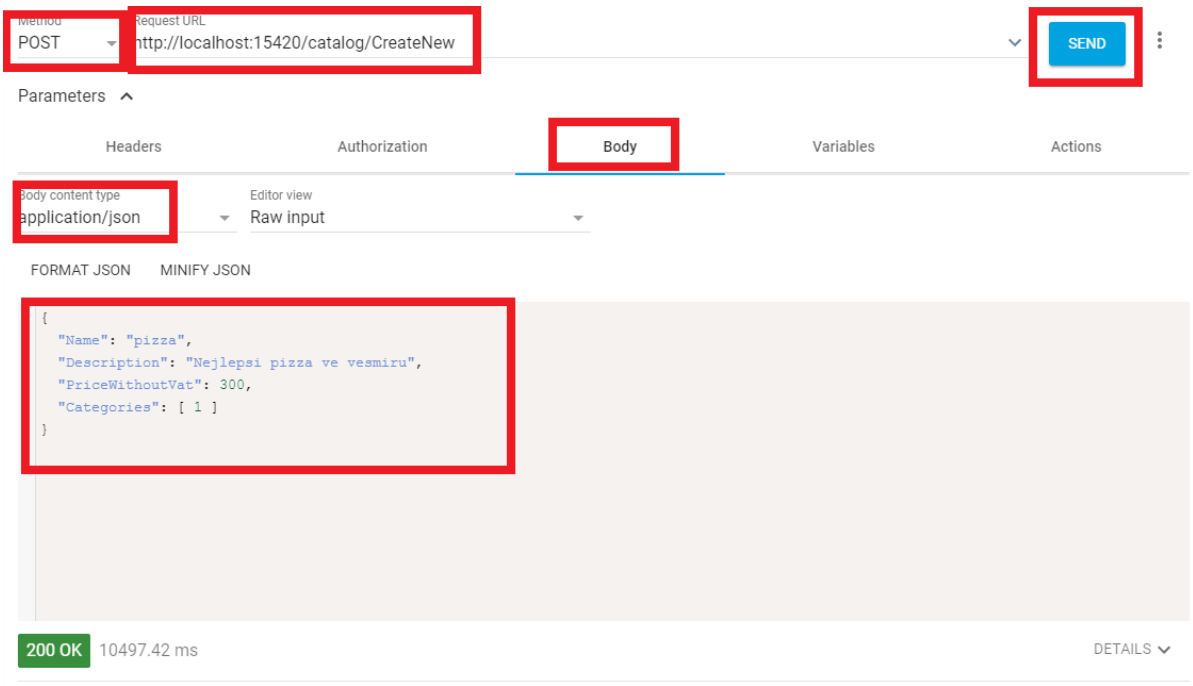
10. V projektu Web rozšiřte CatalogController o další action metodu, která bude přijímat nově vytvořené ProductDto a zavolá náš mediátor (je třeba instnaci ProductDto převést na instnaci NewProductMessage – zatím jen jednoduchým přiřazením všech hodnot).

```
[HttpPost]
public async Task<ActionResult> CreateNew( [FromBody] ProductDto product)
{
    if (ModelState.IsValid)
    {
        var message = new NewProductMessage
        {
            Categories = product.Categories,
            Name = product.Name,
            Description = product.Description,
            PriceWithoutVat = product.PriceWithoutVat
        };
        var response = await MediatorsInfrastructure.Send(message);
        if (response.Success)
            return Ok();
        return BadRequest();
    }
    else
    {
        return BadRequest(ModelState);
    }
}
```

11. Otestujte novou API metodu odesláním POST požadavku na /catalog/CreateNew s následujícím JSON tělem:

```
{
 "Name": "pizza",
 "Description": "Nejlepsi pizza ve vesmiru",
 "PriceWithoutVat": 300,
 "Categories": [ 1 ]
}
```

12. Pokud nemáte žádný oblíbený REST klient, lze použít AdvancedRestClient, jehož instalačka je k dispozici na disku L: . Další oblíbený klient je PostMan, který na okolo vyžaduje registraci, ale dole je možnost „skip registration“. Screenshot z ARC, kde je vidět nastavení



9. Mapování DTO pomocí AutoMapperu

Veškeré změny budou pouze v projektu Web.

1. Nainstalujte do projektu nuget **AutoMapper**.
2. Vytvořte konfigurační třídu **MapperConfig**, která inicializuje všechny mapy (tedy tu jednu, co tam máme)

```
public static class MapperConfig
{
	public static void InitAutomapper()
	{
		Mapper.Initialize(cfg =>
		{
			cfg.CreateMap<ProductDto, NewProductMessage>();
		});
	}
}
```

3. Metoda InitMapper musí být zavolána při sttartu aplikace, proto do hlavní třídy aplikace v **Startup**, do metody **ConfigureServices** přidejte volání metody MapperConfig.InitAutomapper().
4. V **CatalogController** v metodě CreateNew musíme změnit původní ruční mapování z DTO na message na automatické mapování pomocí AutoMapperovské funkce Mapper.Map<NewProductMessage>(product). Metoda tak bude vypadat následovně:

```
[HttpPost]
public async Task<ActionResult> CreateNew( [FromBody] ProductDto product)
{
    if (ModelState.IsValid)
    {
        var message = Mapper.Map<NewProductMessage>(product);
        var response = await MediatorsInfrastructure.Send(message);
        if (response.Success)
            return Ok();
        return BadRequest();
```

```
            }
            else
            {
                return BadRequest(ModelState);
            }
        }
```

5. Otestujte, že aplikace funguje v pořádku a nikde nepadá.

Bonus úloha: podobně jde AutoMapper využít i v NewProductHandler pro mapování z message na Model. Bude nicméně třeba již zapojit konfigurační nastavení. Nejdřív bude potřeba na ProductCategoryRepository přidat metodu, která z DB vytáhne více položek podle jejich ID (public List<ProductCategory> GetMultiple(List<int> ids)) a potom ji použít v mappingu. Pro úplnost upravený MapperConfig

```
public static class MapperConfig
{
	public static void InitAutomapper()
	{
		Mapper.Initialize(cfg =>
		{
			cfg.CreateMap<ProductDto, NewProductMessage>();
			cfg.CreateMap<NewProductMessage, Product>()
				.ForMember(x => x.Categories, x => x.ResolveUsing(m
=> GetCategories(m.Categories)))
				.ForMember(x => x.Id, x => x.Ignore());
		});
	}

	static List<ProductCategory> GetCategories(List<int> ids)
	{
		var db = new ShopDbContext();
		var repository = new ProductCategoryRepository(db);
		return repository.GetMultiple(ids);
	}
}
```

10. Nákupní košík

Cílem cvičení je vytvořit nákupní košík – buď zjednodušeně; nebo bonusová úloha je pak ho vytvořit se vším všudy ☺. Košík budeme uchovávat v session proměnné s názvem „cart“. Přístup k objektu je tedy pomocí funkce `HttpContext.Session.GetString("cart")` .

1. Do projektu web nainstalujte nuget **`Microsoft.AspNetCore.Session`**
2. Do **Startup** do metody ConfigureServices přidejte řádek:
   `services.AddSession();`
3. Do **Startup** do metody Configure přidejte řádek nad UseMvc:
   `app.UseSession();`
4. Vytvořte nový controller CartController, který bude sloužit k obsluze věcí okolo košíku.
5. Vytvořte v něm akci AddToCart, která bude přijímat parametr s ID produktu, které chceme přidat do košíku.
   a. Akce buď může přijímat ID produktu jako int parametr (jednodušší) nebo lze vytvořit model, který by ale obsahoval jen toto ID (obecně lepší).
   b. V OrderService jsme dělali metodu AddToCart, kterou nyní můžeme s výhodou využít na přidání dat do našeho košíku. Koším načítáme a ukládáme do Session.

   c. Po uložení produktu do košíku je dobré uživatele přesměrovat (`return RedirectToAction("ThankYou");`) na „děkovací“ akci, té je dobré předat ID produktu, který byl přidán. Toho lze zařídit pomocí TempDat: `TempData["product"] = productId;`
   d. Bonusová úloha: akce může přijímat i parametr Quantity s počtem produktů, které uživatel chce přidat do košíku.
6. Vytvořte v něm akci ThankYou, která bude sloužit pro zobrazení stránky „zboží bylo přidáno, díky“. Tato akce si vyčte data z TempData, která tam uložil předchozí požadavek. Zobrazíme jednoduché View, která jen vypíše „zboží XY bylo přidáno, díky“. Volitelně s odkazem na košík (dlaší bod).
7. Vytvořte v něm akci Show, která zobrazí obsah košíku. Obsah košíku máme uložen v session proměnné „cart“ (pozor – může být null). V košíku jsou ale jen ID produktu, proto si vytvoříme metodu **OrderService.CreateCartViewModels**, která vezme náš košík a vytvoří z něj pole viewmodelů, kde bude počet produktů v košíku, jeho název atp.
   a. Jedna z možných signatur této služby: `public List<CartItemViewModel> CreateCartViewModels(Cart cart)`
   b. Tato služba si z DB vytáhne seznam produktů dle ID. Buď je můžeme stahovat po jednom (neefektivní, ale v našich počtech je to jedno) nebo na repository produktů implementovat GetByIds (efektivnější, v našich počtech je to jedno).
   c. Pokročilá bonusová úloha: můžete zkusit implementovat GetByIds na bázové repository, což obnáší dynamické generování Expression tree.
   d. View Cart\Show.cshtml bude mít model List<CartItemViewModel> a zobrazí tabulku produktů v košíku.
8. Upravte view Catalog\Category.cshtml tak, aby zobrazoval odkaz na naši novou akci CartController.AddToCart, tedy na přidání produktu do košíku.
   a. Hint: `@Html.ActionLink("Přidat do košíku", "AddToCart", "Cart", new { productId = p.Id }, new { @class = "btn btn-default" })`
   b. Bonusová úloha: lze implementovat jednoduchý formulář, který umožní i zadání počtu
9. Otestujte
10. Bonusová úloha: lze implementovat i stránku pro vytvoření objednávky z košíku – službu pro vytvoření objednávky z košíku jsme implementovali minulé cvičení.
11. Ukázkové cvičné řešení je jako obvykle k dispozici na L:\MvcDemoShop10\netcore

11. Jednoduchý obecný MVC grid

Ukázkové cvičné řešení je jako obvykle k dispozici na L:\MvcDemoShop11\netcore

Ve cvičení jsou některé netriviální kousky kódu, např. GridViewHelper, které je lepší převzít z cvičného řešení.

1. Nejdřív musíme vytvořit potřebné view modely, např takto:

```
public class GridViewParams
{
        public const int PageSize = 10;
        public int Page { get; set; }
        public string Sort { get; set; }
}

public class GridViewModelBase<TEntity>
```

```
{
        public List<TEntity> Data { get; set; }
        public int TotalPages { get; set; }
        public int TotalRecords { get; set; }
        public GridViewParams Parameters { get; set; }
        }
```

2. Dále je třeba vytvořit pomocné služby pro automatické aplikování parametrů gridu – třídu GridViewHelpers (měla by být v BusinessLayer). Tato třída bude mít jednu metodu, která vezme zdroj dat (v našem případě IEnumerable, v praxi asi spíše IQueryable, na což nemáme hotovou infrastrukturu) a GridViewParams a aplikuje stránkování (LINQ metody Skip a Take) a řazení (magie s expressions a voláním generických metod pomocí reflexe – viz příklad).
3. Dále je potřeba vytvořit PagedCategoryDetailViewModel, který je obdobou prostého CategoryDetailViewModel, akorát je odvozen z GridViewModelBase<ProductListViewModel>. Navíc má pouze property string Name (jako CategoryDetailViewModel).
4. Dále vytvořte novou metodu na službě ProductCategoryService: `PagedCategoryDetailViewModel GetProductsByCategoryPaged(int categoryId, GridViewParams parameters)`. Tato metoda by měla dělat cca to samé jako již hotová GetProductsByCategory, ale aplikuje na ni řadící a stránkovací parametry pomocí GridViewHelpers (vytvořené v předchozím bodě).
5. Na CatalogControlleru vytvořte novou akci `CategoryPaged(int id, GridViewParams gridViewParams)`, která bude sloužit jako obdoba akce Category, akorát bude volat Paged variantu metody GetProductsByCategory.
6. K tomuto controlleru vytvořte i View `Catalog\CategoryPaged.cshtml`, které bude mít model PagedCategoryDetailViewModel a zobrazí tabulku.
7. Volitelně: tabulku s produkty přesuňte do samostatného parciálního view `_ProductsGrid.cshtml`, které potom vyrenderujte v `Catalog\CategoryPaged.cshtml` – na to slouží HTML helper `@Html.Partial("_ProductsGrid")`.
8. Zkuste zavolat naší novou akci s novými parametry Page a Sort, tedy např `/Catalog/CategoryPaged?Sort=Name` nebo `/Catalog/CategoryPaged?Page=1`.
9. Bonusová úloha: náš grid nemá klikadla na řazení a stránkování ☺ Můžete tedy implementovat odkazy pro řazení (asi by se měl nacházet v hlavičce) a stránkovač (umožní přejít na stránku 1, 2, 3, … dle počtu záznamů).

12. Zavedení DI kontejneru do aplikace a nastavení lifestyle

1. V tomto cvičení přizpůsobíme služby programování pomocí DI a závislostí.
2. Nejdřív je třeba zkorigovat závislosti tak, aby odpovídaly realitě:
    a. OrderService by měla mít závislost ne na DB Contextu, ale na jednotlivých repository, tj. IProductRepository, IOrderRepository a IUnitOfWork, tedy její konstruktor by měl vypadat:

```
public OrderService(IProductRepository productRepository,
RepositoryBase<Order> orderRepository, IUnitOfWork uow)
{
        this.productRepository = productRepository;
        this.orderRepository = orderRepository;
        this.uow = uow;
}
```

   b. ProductCategoryService by měla obdobně záviset na IProductCategoryRepository a IProductRepository.
   c. CacheService žádné závislosti nemá.
3. Dále vytvořte IOrderService pomocí refactoringu Extract Interface z OrderService, tedy bude mít metody AddToCart, CreateOrder a CreateCartViewModels.
4. Vytvořte „markovací" IService, které nechte prázdné. Tento interface „naimplementujte" na všech našich službách, tj OrderService, ProductCategoryService a CacheService.
5. Tím máme hotové služby a můžeme se pustit do controllerů.
   a. CartController by měl záviset na IOrderService, jeho konstruktor tedy bude vypadat:

```
private readonly IOrderService orderService;
public CartController(IOrderService orderService)
{
        this.orderService = orderService;
}
```

   b. Upravte jednotlivé akce tak, aby používali instanční field orderService namísto ručně vytvářené instance OrderService (a smažte vytváření DB Contextu, který je nyní k ničemu).
   c. CatalogController by měl záviset na IProductCategoryService.
   d. Upravte jednotlivé akce tak, aby používali instanční field productCategoryService namísto ručně vytvářené instance ProductCategoryService (a smažte vytváření DB Contextu, který je nyní k ničemu).
6. Projekt by nyní měl jít zkompilovat, ačkoliv ne spustit kvůli chybějícím bezparametrickým konstruktorům controllerů.
7. Nainstalujte Nuget **Castle.Windsor** a **Castle.Windsor.MsDependencyInjection** do projektu Web.
8. Nejdříve je třeba vytvořit installer, tedy třídu implementující IWindsorInstaller někde v projektu Web. A v ní:
   a. Zaregistrujte náš UnitOfWork pod rozhraním IUnitOfWork.

```
container.Register(Component.For<IUnitOfWork>().ImplementedBy<UnitOfWork>().LifestyleTransient());
```

   b. Hromadně zaregistrujte všechny IService, které se nachází v assembly BusinessLayer (ta se nejjednodušeji identifikuje podle typu, který obsahuje, tj třeba IService.

```
container.Register(Classes.FromAssemblyContaining<IService>().BasedOn<IService>().WithServiceSelf().WithServiceDefaultInterfaces().LifestyleTransient());
```

   c. Dtto s našimi repository, nicméně zde je použít lehce jinou syntax, neboť IRepository je generický interface. Windsor naštěstí umí vyhledávat i podle oteřeného generického typu (open generic type – tj. generický typ bez zadaného typového parametru).

```
container.Register(Classes.FromAssemblyContaining<IProductRepository>().BasedOn(typeof(IRepository<>)).WithServiceSelf().WithServiceDefaultInterfaces().LifestyleTransient());
```

9. Nyní je třeba nastavit náš container – v **Startup** v metodě **ConfigureServices**:
   a. Vytvořte instanci WindsorContainer
      `var windsorContainer = new WindsorContainer();`
   b. Spustťe vytvořený instalátor
      `windsorContainer.Install(new WindsorInstaller());`
   c. Upravte metodu ConfigureServices tak, aby vracela IServiceProvider
   d. Vytvořte a vraťte integrační service provider:
      `return WindsorRegistrationHelper.CreateServiceProvider(windsorContainer, services);`

10. Spusťte aplikaci a měla by fungovat v pořádku.

13. Windsor – cache interceptor

V tomto cvičení vytvoříme interceptor (automaticky implementovaný vzor Dekorátor, který bude cachovat výstup metod označených atributem CacheMe.

1. Nejdřív musíme vytvořit atribut CacheMe, který bude sloužit hlavně jako marker.

```
public class CacheMeAttribute : Attribute
{
        public CacheMeAttribute(int durationInMinutes = 60)
        {
                DurationInMinutes = durationInMinutes;
        }
        public int DurationInMinutes { get; set; }
}
```

2. Dále je třeba nainstalovat Nuget Castle.Core do projektu BusinessLayer, neboť potřebujeme rozhraní IInterceptor.
3. Vytvořte vlastní interceptor CacheInterceptor implementující rozhraní IInterceptor. Toto má jednu metodu Intercept s parametrem typu IInvocation:
    a. Property Method uvádí metodu, která je spouštěna.
    b. Property TargetType uvádí typ, kde je ona metoda.
    c. Metoda Proceed spustí další interceptor v řadě nebo původní metodu.
    d. Property ResultType obsahuje návratovou hodnotu volané metody – buď po zavolání Proceed, nebo ji můžeme nastavit ručně. Když pak vynecháme volání Proceed, tak se metoda vůbec nespustí.
    e. Náš interceptor bude mít závislost na ICacheService.
4. Metoda intercept bude fungovat následovně:
    a. Pokud na invocation.Method neexistuje atribut CacheMeAttribute (zjistíme `invocation.Method.GetCustomAttribute<CacheMeAttribute>())`, tak voláme Proceed a konec.
    b. Nejdřív sestavíme cachovací klíč: `var key = invocation.TargetType.FullName + "." + invocation.Method.Name;`
    c. Následně pomocí cacheService zjistímee, jestli máme hodnotu nacachovanou.
    d. Pokud ano, tak ji nastavíme do ReturnValue a konec.
    e. Pokud ne, tak zavoláme Proceed, čímž se nám objeví výsledná hodnota v property ReturnValue. Tu můžeme pomocí CacheService uložit do cache.
5. Musíme zbavit CacheService rozhraní IService a zaregistrovat ji ručně kvůli cyklu, který by nám jinak vznikl.
6. Upravíme registraci komponent pomocí Windsor následovně (ve třídě **WindsorInstaller)**:
    a. Ruční registrace naší ICacheService a nového interceptoru.
    ```
    container.Register(Component.For<ICacheService>().ImplementedBy<CacheService>().LifestyleTransient());
    container.Register(Component.For<CacheInterceptor>().LifestyleTransient());
    ```
    b. Je třeba nastavit interceptor pro všechny služby, což lze udělat vytvořením metody:

```
private void ConfigureInterceptors(ComponentRegistration obj)
{
var reg = obj.Interceptors(InterceptorReference.ForType<CacheInterceptor>()).First;
```

```
}
```

c. Tato metoda nakonfiguruje službu, která se ji předá. Je třeba takto nakonfigurovat všechny služby, takže k registraci našich byznys služeb (IService) přidejte volání `.Configure(ConfigureInterceptors)`. Registrace pak bude vypadat:

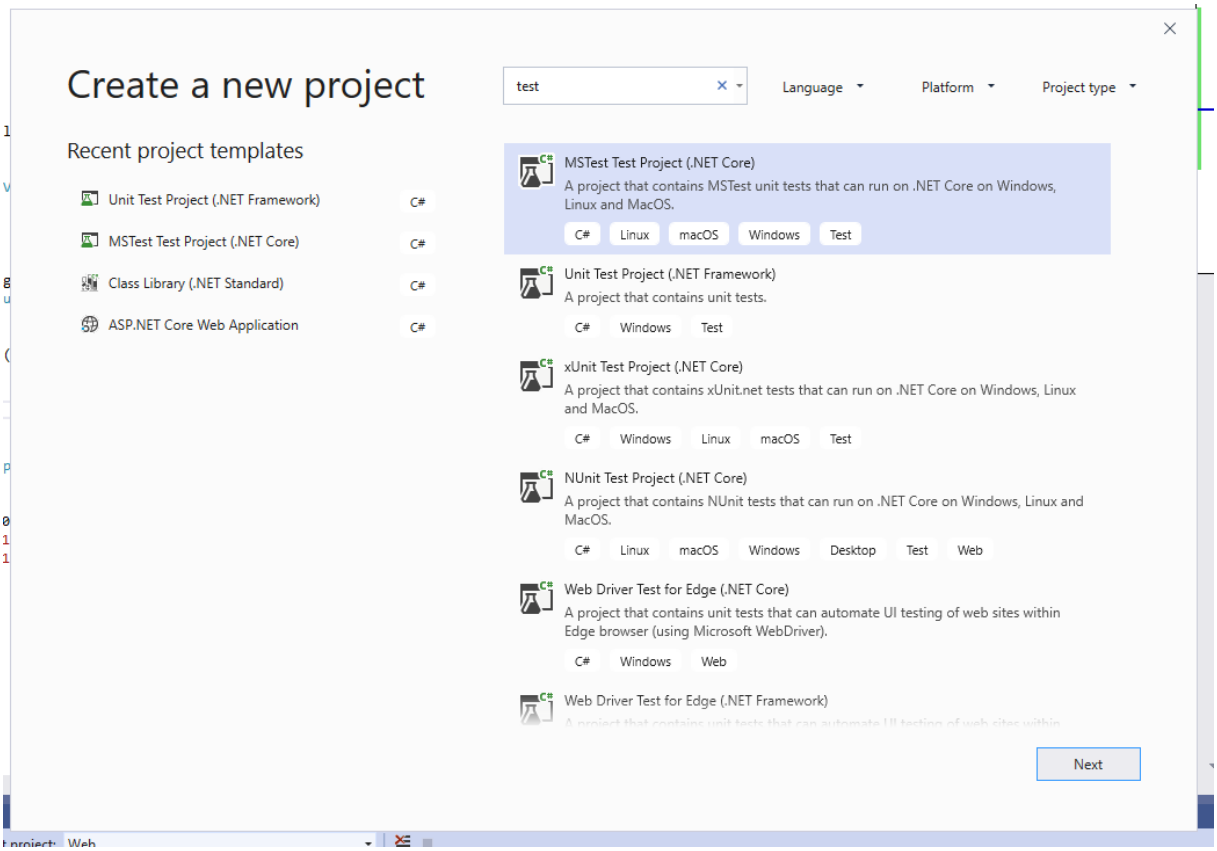
```
container.Register(Classes.FromAssemblyContaining<IService>().BasedOn<IService>().WithServiceSelf().WithServiceDefaultInterfaces().LifestyleTransient().Configure(ConfigureInterceptors));
```

7. Přidejte atribut [CacheMe] na metodu IProductCategoryService.GetRootCatalog (POZOR – na interface)
8. Otestujte zobrazením seznamu kategorií, měl by se volat interceptor a probíhat cachování.

14. Základní unit testy

Cílem cvičení je vytvořit několik testů na metodu GridViewHelper.Apply z minulých cvičení.

1. Vytvořte nový MSTest Test Project (.NET Core) jménem „Test“ (na jméně nezáleží).



2. V projektu se již vytvoří základní šablona našeho prvního testu.
3. Vytvořte několik unit testů na testování správnosti metody – různé kombinace parametrů atp.

Jedním z možných je např:

1. Vytvoříme testovací data

```
var parameters = new GridViewParams()
```

```
{
        Page = 1,
        Sort = "Id"
};
var data = Enumerable.Range(0, 30)
       .Select(i => new ProductListViewModel
       {
              Id = i,
              Name = i.ToString(),
       })
       .Reverse()
       .ToList();
```

2. Zavoláme metodu ApplyParams

```
var applied = GridViewHelpers.ApplyParams(parameters, data);
```

3. Ověříme výsledek:

```
Assert.AreEqual(10, applied.Count);
Assert.AreEqual("10", applied[0].Name);
Assert.AreEqual("19", applied[9].Name);
```

4. Spusťte test v Test Exploreru. Toto okno lze vyvolat nabídkou Test->Windows->Test Explorer. Je třeba zkompilovat celé solution než se testy zobrazí.

Další možný test:

1. Změňte v předchozím testu property Sort na „liška“, smažte všechny asserty a spusťte test.
2. Chceme, aby test spadnul např. na `InvalidOperationException`. Takže odekorujeme testovací metodu navíc atributem `[ExpectedException(typeof(InvalidOperationException))]`
3. Test spadne, což je dobře, ale na špatnou výjimku, což je špatně.
4. Opravte metodu `ApplyParams tak`, aby při chybně zadané property `Sort` vyvolala výjimku `InvalidOperationException`.
5. Ověřte, že vše je OK.
6. Gratuluji, právě jste zažil/a Test Driven Development v praxi.

15. Mockujeme testy se závislostmi pomocí Moq

1. Nainstalujte Nuget **Moq** do projektu Test.
2. Vytvořte novou test metodu GetRootCatalog, která bude testovat ProductCategoryService.GetRootCatalog.
3. Pomocí mockování vytvořte mockovací implementace rozhraní IProductRepository a IProductCategoryRepository – stačí mockovat IProductCategoryRepository.GetAll, která vrací List<ProductCategory>.
4. Vytvořte instanci testovací třídy a zavolejte metodu GetRootCatalog.
5. Ověřte výsledek.
6. Zkuste další testy na jiné metody ☺

Příklad unit testu na metodu GetRootCatalog:

```
[TestMethod]
public void GetRootCatalog()
{
       // vytvoříme mocky
```

```
        Mock<IProductRepository> productRepositoryMock = new Mock<IProductRepository>();
        Mock<IProductCategoryRepository> productCategoryRepositoryMock = new
Mock<IProductCategoryRepository>();
        // namockujeme metodu GetAll aby vracela nějaká testovací data
        productCategoryRepositoryMock.Setup(x => x.GetAll()).Returns(new
List<ProductCategory>() { new ProductCategory() { Name = "XY" } });

        // vytvoříme testovací třídu
        var service = new ProductCategoryService(productCategoryRepositoryMock.Object,
productRepositoryMock.Object);

        // spustíme testovanou metodu
        var result = service.GetRootCatalog();

        // ověříme výsledek
        Assert.AreEqual(1, result.Categories.Count);
        Assert.AreEqual("XY", result.Categories[0].Name);
}
```