Entity Framework – ORM – Code First подход

Създаване, свързване на база данни в ASP .NET Core проекти

1. Създайте обикновен ASP .NET Core проект.
2. Нужно е инсталация на няколко пакета на EntityFramework (EF)

Десен бутон на проекта в SalutionExplorer -> Manage NuGet Packages-> в Browse търсим:

* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.Tools

и ги инсталираме.

1. Създайте папка **Data** в директорията на проекта. А в нея още една папка **Models** (тук ще създаваме моделите за базата данни) и един клас ShoppingListDbContext.cs, който трябва да наследява DbContext от EF. В него дефинираме конструктора:

public ShoppingListDbContext(DbContextOptions<ShoppingListDbContext> options)

: base(options) { }

Параметърът **options** съдържа конфигурационни настройки за контекста на базата данни, като доставчика на базата данни (например SQL Server, SQLite) и низ за връзка (connection string).

1. Закачаме проекта за базата:
   1. В appsettings.json добавяме нашия ConnectionString:

"ConnectionStrings": {

"DefaultConnection": "Server=(LocalDB)\\MSSQLLocalDB;Database=ShopingListDb;User Id=; Password=; TrustServerCertificate=True;"

},

Където пишем име на връзката (DefaultConnection), задаваме името на сървъра Server=, името на базата данни (в случая ShopingListDb ), ако имаме user id и парола и задължително накрая TrustServerCertificate=True.

* 1. Следваща стъпка е да свържем IoC контейнера с DbContext-а.

В Program.cs задължително преди инстанцирането на нашето приложение ( преди var app = builder.Build(); ) поставяме следните два реда:

string connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<ShoppingListDbContext>(opt => opt.UseSqlServer(connectionString));

1. В папка Data/Models създаваме клас Product.cs, където свойствата му ще бъдат колоните на таблицата Products в базата ни данни. Задължително е да извикаме System.ComponentModel.DataAnnotations библиотеката за да пишем коментари и да задаваме атрибути на свойствата като [Key], [Required], [ForeignKey], уточнявайки първичния ключ, задължителни полета, външни ключове, лимити, правила за валидация, форматиране и други изисквания към данните.

[Comment("Shopping List Product")]

public class Produc

{

[Key]

[Comment("Product Indentifier")]

public int Id { get; set; }

[Required]

[MaxLength(100)]

public string Name { get; set; }=string.Empty;

public List<ProductNote> ProductNotes { get; set; } = new List<ProductNote>();

}

1. В папка Data/Models създаваме клас ProductNote.cs . Това ще е втората ни таблица.

public class ProductNote

{

[Key]

[Comment("Note Indentifier")]

public Guid Id { get; set; }

[Required]

[Comment("Note Content")]

public string Content { get; set; } = string.Empty;

[Required]

[Comment("Product Indentifier")]

public int ProductId { get; set; }

[ForeignKey(nameof(ProductId))]

public Product Product { get; set; } = null!;

}

1. Дописваме свойства на класа ShoppingListDbContext.cs, които служат за представяне на таблици в базата данни:

public DbSet<Product> Products { get; set; }

public DbSet<ProductNote> ProductNotes { get; set; }

1. Следваща стъпка е да създадем базата данни. За целта отваряме

View -> Other Windows -> Package Manager Console и пишем командите:

* add-migration InitialMigration ,

която създава нова миграция за базата данни въз основа на текущия модел на данните в приложението.

* update-database,

която прилага промените към базата ни данни.

1. В класа ShoppingListDbContext.cs чрез метода OnModelCreating(……) ще създадем два записа в таблица Products:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Product>()

.HasData(

new Product() { Id = 1, Name = "Cheese" },

new Product() { Id = 2, Name = "Milk" }

);

base.OnModelCreating(modelBuilder);

}

OnModelCreating се извиква автоматично при създаване на контекста на базата данни и е идеалното място за настройка на сложни релации между таблиците.

С командите add-migration SeedProducts и update-database актуализираме данните в базата.

1. Dependency Injection (DI)
   1. В Models създаваме нов клас ProductViewModel.cs, където описваме модела и залагаме верификация на въвежданите данни на потребителско ниво.

/// <summary>

/// Product model

/// </summary>

public class ProductViewModel

{

/// <summary>

/// Product Indentifier

/// </summary>

public int Id{ get; set; }

/// <summary>

/// Product Name

/// </summary>

[Required(ErrorMessage ="Field {0} is required")]

[Display(Name ="Product Name")]

[StringLength(100, MinimumLength = 3, ErrorMessage ="Field {0} must be between {2} and {1} symbols")]

public string Name { get; set; }=string.Empty;

}

* 1. Ще отделим бизнес логиката на приложението в отделен сървиз. За целта създаваме нови две папки в директорията на проекта:
* папка Contracts и в нея нов интерфейс IProductService.cs, където се залагат асинхронни методи, които ще извличат заложените от ProductViewModel типове данни.

public interface IProductService

{

Task<IEnumerable<ProductViewModel>> GetAllAsync();

Task<ProductViewModel> GetByIdAsync(int id);

Task AddProductAsync(ProductViewModel model);

Task UpdateProductAsync(ProductViewModel model);

Task DeleteProductAsync(int id);

}

* папка Services, а в нея нов клас ProductService.cs, който наследява IProductService и ще имплементира методите му.

public class ProductService : IProductService

{

private readonly ShoppingListDbContext context;//частно поле

public ProductService(ShoppingListDbContext \_context)//конструктор

{

context= \_context;

}

Конструктор на клас ProductService, който инжектира зависимост от типа ShoppingListDbContext и я запазва в частно поле на класа. Dependency Injection (DI) позволява на ProductService да взаимодейства с базата данни, като използва контекста за извличане, добавяне, актуализиране и изтриване на данни.

* 1. Създаваме нов контролер в папка Controllers-> ProductController.cs;

private readonly IProductService productService;

public ProductController(IProductService \_productService)

{

productService = \_productService;

}

Конструктор в контролер ProductController, който използва Dependency Injection, за да инжектира зависимост от интерфейса IProductService. Този подход позволява на контролера да използва методи от IProductService за взаимодействие с данни, свързани с продукти, без самият контролер да трябва да знае как е реализирана логиката за достъп до базата данни.

* 1. Добавяне на сървиза към IoC контейнера:

В Program.cs задължително преди инстанцирането на нашето приложение ( преди var app = builder.Build(); ) поставяме следния ред:

builder.Services.AddScoped<IProductService, ProductService>();

1. CRUD (Create, Read, Update, Delete) операции
   1. Добавяне на изглед Index.cshtml в папка View/ Product

@model List<ProductViewModel>

@{

ViewBag.Title = "All Products";

}

<h2 class="text-center">@ViewBag.Title</h2>

<hr />

<div class="d-grid gap-2 mb-2">

<**a** **asp-controller**="Product" **asp-action**="Add" class="btn btn-primary">Add Product</**a**>

</div>

@if (Model.Count() > 0)

{

<div class="row">

@foreach (var product in Model)

{

<div class="col-sm-6">

<div class="card">

<div class="card-body">@product.Name

<div class="d-grid gap-2">

<**a** **asp-controller**="Product" **asp-action**="Edit" **asp-route-id**="@product.Id" class="btn btn-warning">Edit</**a**>

<**form** class="mt-2" **asp-controller**="Product" **asp-action**="Delete" **asp-route-id**="@product.Id">

@Html.AntiForgeryToken()

<div class="d-grid gap-2">

<input type="submit" value="Delete" class="btn btn-danger mb-2" />

</div>

</**form**>

</div>

</div>

</div>

</div>

}

</div>

}

else

{

<p class="text-center">No products yet!</p>

}

* 1. В ProductService.cs добавяме:

public async Task<IEnumerable<ProductViewModel>> GetAllAsync()

{

return await context.Products

.AsNoTracking()

.Select(p => new ProductViewModel()

{

Id = p.Id,

Name = p.Name

})

.ToListAsync();

}

* 1. Добавяме Get Index действиe (action) за визуализиране на всички продукти в ProductController.cs:

public async Task<IActionResult> Index()

{

var model = await productService.GetAllAsync();

return View(model);

}

* 1. Добавяне на изглед Add.cshtml в папка View/ Product

@model ProductViewModel

@{

ViewBag.Title = "Add Product";

}

<h2 class="text-center">@ViewBag.Title</h2>

<hr />

<div class="row">

<**form** method="post">

<div class="form-group">

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" placeholder="Write your product here...." />

<**span** **asp-validation-for**="Name" calss="samll text-danger"></**span**>

</div>

</div>

<input class="btn btn-primary mt-3" type="submit" value="Create" />

</**form**>

</div>

@\*

Валидация

\*@

@section Scripts{

<**partial** **name**="\_ValidationScriptsPartial" />

}

* 1. В ProductService.cs добавяме:

public async Task AddProductAsync(ProductViewModel model)

{

var entity = new Product()

{

Name = model.Name

};

await context.Products.AddAsync(entity);

await context.SaveChangesAsync();

}

* 1. В ProductController.cs пишем Get и Post методи за Add действието (action) за добавяне на продукт е базата данни:

[HttpGet]

public IActionResult Add()

{

var model = new ProductViewModel();

return View(model);

}

[HttpPost]

public async Task<IActionResult> Add(ProductViewModel model)

{

if(ModelState.IsValid == false)

{

return View(model);

}

await productService.AddProductAsync(model);

return RedirectToAction(nameof(Index));

}

* 1. Добавяне на изглед Edit.cshtml в папка View/ Product

@model ProductViewModel

@{

ViewBag.Title = "Add Product";

}

<h2 class="text-center">@ViewBag.Title</h2>

<hr />

<div class="row">

<**form** method="post">

<div class="form-group">

<**input** **asp-for**="Id" **type**="hidden" />

<div class="mb-3">

<**label** **asp-for**="Name"></**label**>

<**input** **asp-for**="Name" class="form-control" placeholder="Write your product here...." />

<**span** **asp-validation-for**="Name" calss="samll text-danger"></**span**>

</div>

</div>

<input class="btn btn-primary mt-3" type="submit" value="Edit" />

</**form**>

</div>

@\*

Валидация

\*@

@section Scripts {

<**partial** **name**="\_ValidationScriptsPartial" />

}

* 1. В ProductService.cs добавяме:

public async Task<ProductViewModel> GetByIdAsync(int id)

{

var entity = await context.Products.FindAsync(id);

if(entity == null)

{

throw new ArgumentException("Invalid product");

}

return new ProductViewModel()

{

Id = id,

Name = entity.Name

};

}

public async Task UpdateProductAsync(ProductViewModel model)

{

var entity = await context.Products.FindAsync(model.Id);

if (entity == null)

{

throw new ArgumentException("Invalid product");

}

entity.Name = model.Name;

await context.SaveChangesAsync();

}

* 1. В ProductController.cs пишем Get и Post методи за Edit действието (action) за добавяне на продукт е базата данни:

[HttpGet]

public async Task<IActionResult> Edit(int id)

{

var model = await productService.GetByIdAsync(id);

return View(model);

}

[HttpPost]

public async Task<IActionResult> Edit(ProductViewModel model, int id)

{

if(ModelState.IsValid == false || model.Id != id)

{

return View(model);

}

await productService.UpdateProductAsync(model);

return RedirectToAction(nameof(Index));

}

* 1. В ProductService.cs добавяме метод за премахване на продукт от базата данни:

public async Task DeleteProductAsync(int id)

{

var entity = await context.Products.FindAsync(id);

if (entity == null)

{

throw new ArgumentException("Invalid product");

}

context.Products.Remove(entity);

await context.SaveChangesAsync();

}

* 1. В ProductController.cs пишем метод за Delete действието, което изпълнява Post заявка:

[HttpPost]

public async Task<IActionResult> Delete(int id)

{

await productService.DeleteProductAsync(id);

return RedirectToAction(nameof(Index));

}