ASP.NET

By Khaled Frayji

Khaledfrayji007@gmail.com

Contents

Razor pages

➢ Database connectivity:

DbContext:

```
public class DataContext : DbContext
{
    public DataContext(DbContextOptions<DataContext> options):

base(options) {  }

    public DbSet<MyModel> Models { get; set; }
}
```

Connection String:

```
"ConnectionStrings": {
  "conn":
"Server=(localdb)\\mssqllocaldb;Database=DatabaseName;Trusted_Connection=True;MultipleActiveResultSets=true"
 }
```

Registration (program.cs):

```
builder.Services.AddDbContext<DataContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("conn")));
```

Migration:

```
add-migration init
```

Create database:

```
update-database
```

Generate the Scaffolding with one command: (Not allowed in the exam)

dotnet ef dbcontext scaffold "data source=SqlServerName;initial catalog=DatabaseName;integrated security=True;TrustServerCertificate=True" Microsoft.EntityFrameworkCore.SqlServer --context MyDbContext --output-dir Models --table MyModel

➢ Binding Model Class example:

```
public class MyModel

{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public DateTime BirthDate { get; set; }

}
```

How to bind ?

In the Razor Page Model (.cshtml.cs) :

[BindProperty]

Public MyModel model {get;set;}

➢ View Model vs Binding Model Classes:

View and Binding model should have same fields with the same validation (logic).

Use the view model for client side validation.

"

Use the binding model for data binding and server side validation.(use name attribute to specify the field that will bound from).

<input name="Age" asp-for="ViewModel.Age" />

➢ Server side validation:

Use data annotation attributes.

Binding Model:

```
using System.ComponentModel.DataAnnotations;

public class MyModel
{
   [Required(ErrorMessage = "First name is required.")]

   public string FirstName { get; set; }

   [Required(ErrorMessage = "Last name is required.")]

   public string LastName { get; set; }

   [Required(ErrorMessage = "Birth date is required.")]

   [DataType(DataType.Date)]

   [Display(Name = "Birth Date")]
   public DateTime BirthDate { get; set; }
   public int FirstNumber { get; set; }


   public int SecondNumber { get; set; }


   public int Sum { get; set; }

}
```

Page Model:

```
    [BindProperty]
    Public BindingModel m {get;set;}


public IActionResult OnPost()
  {
  if(m.FirstNumber + m.SecondNumber != m.Sum)
              {
                 ModelState.AddModelError(
                    string.Empty, "Verification failed"
                   );

               }

    if (!ModelState.IsValid)

    {
      // Return the page with validation errors
      return Page();
    }

    // Process the data (e.g., save to database)
    // Redirect to a confirmation or success page
    return RedirectToPage("Success");
  }
```

Razor View (.cshtml):

```
<form method="post">

  <div>
    <label asp-for="MyModel.FirstName"></label>
    <input asp-for="MyModel.FirstName" />
    <span asp-validation-for="MyModel.FirstName"></span>
  </div>

  <div>
    <label asp-for="MyModel.LastName"></label>
    <input asp-for="MyModel.LastName" />
    <span asp-validation-for="MyModel.LastName"></span>
  </div>

  <div>
    <label asp-for="MyModel.BirthDate"></label>
    <input asp-for="MyModel.BirthDate" type="date" />
    <span asp-validation-for="MyModel.BirthDate"></span>
  </div>

  <button type="submit">Submit</button>
</form>
```

`<div asp-validation-summary="All"></div>` ⟹ will display server side validations

➢ Client Side Validation:

Add this script somewhere in your view to enable client side validation

```
@section Scripts {
   @Html.Partial("_ValidationScriptsPartial")
}
```

➢ Customizing Error Page:

Create a new Razor page called 404 for example:

```
<h1 style="color:darkred">404 - Page Not Found HAHAHAH</h1>
<p>The page you are looking for does not exist.</p>
```

In program.cs :

```
app.UseStatusCodePagesWithRedirects("/404");
```

➢ Route Parameter:

```
@page "/products/{id:int:min(1):max(1000)}"
```

In this example, {id} is a route parameter that captures the value from the URL. When a request is made to /products/123, the id parameter will be set to 123, and the page will display the details of the product with ID 123 using the Get() method.

Handler keyword:

```
public IActionResult OnGet(int id)
  {
    // Fetch and display product details based on the provided ID
    return Page();
  }

public IActionResult OnGetUpdate(int id)
  {
    // Update product details based on the provided ID
    return RedirectToPage(new { id });
  }
```

Example : /products/123?handler=Update

Handler keyword in the url specifies that OnGetUpdate should be executed:

Note: If you passed the handler by route parameter you access it like the following:

/products/123/Update

The name of handler could be passed :

1. By route parameter : @page "{handler}"
2. By query string parameter
3. By Route or query string parameters : @page "{handler?}"

➢ Inject Service into Razor page using DI:

```
private readonly MyInterface  _i;

public Constructor(MyInterface i){

_i=i;

}
```

➢ Add a Service to the DI Container (program.cs):

```
builder.Services.AddSingleton<MyInterface,MyImplementation>();
```

Minimal API

It is recommended to use IResult as function data type:

Example:

```
public static IResult Square(int number)
 {
    return Results.Ok(number*number);
 }
```

Attributes:

[FromBody] : To bind a json object (complex types) from body.Used with POST, PUT, or PATCH.

[FromQuery]: To bind from a query string. No need for it if you do not receive the parameter from many sources.

[FromRoute]: To bind from the route parameter.

[AsParameters] = To bind multiple parameters from different sources.Used when you have an endpoint that requires parameters from different parts of the request.

Logs to the console:

Console.WriteLine("Added Person : "+ person.ToString());

➢ Validation and Error Handling

Using EndpointFilter:

For a better organization create a new Validation class that contains inner validation classes.

Add the validation to your endpoint like the following example:

app.MapPost("/a",Handler.AddPerson).AddEndpointFilter<Validations.ValidatePerson>();

Example of class :

```
public class Validations {

//inner class to validate person
public class ValidatePerson : IEndpointFilter
 {
    public async ValueTask<object?> InvokeAsync(EndpointFilterInvocationContext context,
EndpointFilterDelegate next)
    {
       var person = context.GetArgument<Person>(0);
       if (person.Id < 1 || person.Id > 1000)
       {
         return Results.ValidationProblem(new Dictionary<string, string[]>
    {
       {"Id", new [] {"Id should be between 1 and 1000"} },
    });
       }
       return await next(context);
    }

    }}
```

Validate within your endpoint:

```
public static IResult GetPerson(int id)

{

        var person = people.FirstOrDefault(x => x.Id == id);

        if (person is null)

                return Results.ValidationProblem(new Dictionary<string, string[]>

                {

                        {"Id", new [] {"Id doesn't exist"} }

                });

        return Results.Ok(person);

}
```

Validation with DataAnnotations:

To enable it you should add **.WithParameterValidation** to your endpoint.

Install the package MinimalApis.Extensions from the NPM to enable **.WithParameterValidation**

Example:

```
app.MapPost("/Add", Handler.AddUser).WithParameterValidation();
```

Validation with IValidatableObject interface:

```
class User : IValidatableObject
{

	public string Username { get; set; }

	public string Email { get; set; }

	public int Age { get; set; }

	public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
	{
		if(Age % 2 != 0)
		{
			yield return new ValidationResult("Age must be multiple
of 2");
		}
		if(!(Email.Contains("gmail.com") ||
Email.Contains("ul.edu.lb")))
		{
			yield return new ValidationResult("Email should contain
either gmail.com or ul.edu.lb");
		}

	}
}
```

BindAsync:

Public class Person{

```
public static async ValueTask<Person?> BindAsync(HttpContext context)
{
       var query = context.Request.Query;
       var person = new Person();

    if (context.Request.RouteValues.ContainsKey("id"))
    {

          if(int.TryParse(context.Request.RouteValues["id"].ToString(),
      out var outId))
           {
                 person.Id = outId;
            }
          else
            {
                 //To indicate that the value is not parsable
                 person.Id = -1;
             }

    }
      else if (query.TryGetValue("id", out var outId))
          {
              if (int.TryParse(outId, out var nId))
              {
                  person.Id = nId;
              }

              else
               {
                 person.Id = -1;

                }
    }

    if (context.Request.RouteValues.ContainsKey("name"))
    {

           person.Name = context.Request.RouteValues["name"].ToString();

    }
    else if (query.TryGetValue("name", out var outName))
    {
           person.Name = outName;
    }



    return person;
}}
```

Explanation:

```
public static IResult GetPersonBindAsync(Person person)
       {
     var results = people.Where(x => x.Id == person.Id || x.Name ==
person.Name).ToList();

     if (results.Count == 0)
     {
         return Results.ValidationProblem(new Dictionary<string, string[]>
 {
     {"Person Not found: ", new [] {"Id and Name do not exist"} }
 });
     }

     return Results.Ok(results);


 }
```

The method parameter is a complexe type so if we use [FromBody] -> BindAsync will not be called.
the same for [AsParameters].
So both they have priority on BindAsync . In our case no attribute specified so BindAsync will be called.

Note : While fetching data ,If you do not specify any attribute and do not implement the BindAsync method and you passed a complexe type as parameter , you will face

An unhandled exception occurred while processing the request.

➢ TryParse:

```
public static bool TryParse(string s, out Product result)
{
        result = default;
        if (decimal.TryParse(s, out decimal price))
        {
                result = new Product()
                {
                        Price = price
                };
                return true;
        }
        return false;
}
```

```
public static IResult AddProductTryParse(string title, string description,Product? product)
```

Because of TryParse method -> Product will not be bound from body but from query parameter.

➢ Implementing CRUD (Create, Read, Update, Delete) operations:

Reminder: It is recommended to use another class (with same attributes) to bind the data than the main class used in database.

Steps:

1. Create Your Database.
2. Create Service Folder and Interface(s).
3. Create Concrete Class(es): These classes will contain the actual logic for accessing and modifying database data.
4. Inject DataContext in Concrete Class Using Dependency Injection.
5. Register Interface and Concrete Class in DI Container in Program.cs.

Functions Example:

Note : It is recommended to use Task<IResult> when implementing CRUD.

Create :

```
public async Task<IResult> CreateProduct(Product p)
        {
                var newProduct = new Product
                {
                        Name = p.Name,
                        Price = p.Price,
            CategoryId = p.CategoryId,
             Category = null, //this is a navigation property.
                };

                await _context.AddAsync(newProduct);
                await _context.SaveChangesAsync();
                return Results.Created($"/products/{newProduct.ProductId}", newProduct);

        }
```

Get All:

```
public async Task<IResult> GetAllProducts()
{
   return Results.Ok(await _context.Products.ToListAsync());
}
```

Get By Id:

```
public async Task<IResult> GetProductById(int id)
 {

    var product = await _context.Products.Include(x => x.Category).FirstOrDefaultAsync(x =>
x.ProductId == id);


    if (product is null)
    {
       return Results.NotFound("Product Not Found");
    }
    return Results.Ok(product);
 }
```

Update:

```
 public void UpdateProduct(Product product, string name,decimal price, int categoryId)
 {
     product.Name = name;
     product.Description = description;
     product.Price = price;
     product.CategoryId = categoryId;
 }


public async Task<IResult> UpdateProduct(int id, Product p)
{
    var product = await _context.Products.FirstOrDefaultAsync(x => x.ProductId == id);
    if (product is null)
    {
       return Results.NotFound("Product Not Found");
    }

    UpdateProduct(product, p.Name, p.Price, p.CategoryId);
    await _context.SaveChangesAsync();
    return Results.Ok(product);
 }
```

Delete:

```
public async Task<IResult> DeleteProduct(int id)
{
        var product = await _context.Products.FirstOrDefaultAsync(x => x.ProductId == id);
        if(product is null)
        {
                return Results.NotFound("Product Not Found.");
        }
        _context.Products.Remove(product);
        await _context.SaveChangesAsync();
        return Results.Ok("Product is deleted.");}
```

Some LINQ aggregate functions:

Count All Products:

```
public async Task<IResult> GetCount()
{
   var count = await _context.Products.CountAsync();
   return Results.Ok(count);
}
```

Sum of price:

```
public async Task<IResult> GetTotalPrice()

{

   var totalPrice = await _context.Products.SumAsync(x => x.Price);

   return Results.Ok(totalPrice);

}
```

Min Price:

```
public async Task<IResult> GetMinPrice()
{
   var minPrice = await _context.Products.MinAsync(x => x.Price);
   return Results.Ok(minPrice);
}
```

Max Price:

```
public async Task<IResult> GetMaxPrice()
{
   var maxPrice = await _context.Products.MaxAsync(x => x.Price);
   return Results.Ok(maxPrice);
}
```

Average:

```
public async Task<IResult> GetAveragePrice()
{
   var count = await _context.Products.CountAsync();
   if (count == 0)
   {
      return Results.Ok(0);
   }
   var average = await _context.Products.AverageAsync(x => x.Price);
   return Results.Ok(average);
}
```

Get First Product:

```
public async Task<IResult> GetFirstProduct()
{
   var firstProduct = await _context.Products.FirstOrDefaultAsync();
   if (firstProduct == null)
   {
      return Results.NotFound();
   }
   return Results.Ok(firstProduct);
}
```

Get Last Product:

```
public async Task<IResult> GetLastProduct()
{
   var lastProduct = await _context.Products.OrderByDescending(p =>
p.Id).FirstOrDefaultAsync();
   if (lastProduct == null)
   {
      return Results.NotFound();
   }
   return Results.Ok(lastProduct);
}
```

➢ Foreign Key Concept in EF:

Considering the following Database model where CourseId is a foreign key in the student table.

Student	Course
StudentID	CourseId
#CourseID	CourseName

Classes implementation:

```
public class Student

{

   public int StudentId { get; set; } //this will be automatically PK.


   // Foreign Key (same name as the course primary key)

   public int CourseId { get; set; }



      // Navigation property This allows you to access the Course related to a Student.
```

```
    public Course? Course { get; set; } //it can be Null.
}
public class Course
{
    public int CourseId { get; set; } //this will be automatically PK.
    public string CourseName { get; set; }
}
```

Supposing this relationship is one to many where each Course can have multiple Students, but each Student can enroll in only one Course.

Add this piece of code to your DbContext class:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
        base.OnModelCreating(modelBuilder);
        modelBuilder.Entity<Student>()
       .HasOne(s => s.Course)
       .WithMany()
       .HasForeignKey(s => s.CourseId);
       .OnDelete(DeleteBehavior.Cascade);
}
```

Explanation:

HasOne(s => s.Course) // Indicates that each Student has one course.
WithMany() //Indicates that each Course can have many Students.

HasForeignKey(s => s.CourseId) // Specifies the foreign key in the Student table.

OnDelete(DeleteBehavior.Cascade) // Configures cascade delete, meaning when a Course is deleted, all related Students are also deleted.

Note: Dr. Kamal has not worked with many-to-many relationships.

➢ Implement CRUD Endpoints Using Minimal APIs:

Create a Product section:

```
var products = app.MapGroup("/products").WithTags("Products");

//Create
products.MapPost("/", async (IProductService productService, Product product)

        => await productService.CreateProduct(product))



//Update
products.MapPut("/{id}", async (IProductService productService, int id, Product product)

        => await productService.UpdateProduct(id, product))


//Delete
products.MapDelete("/{id}", async (IProductService productService, int id)

   => await productService.DeleteProduct(id));
```

Regex

[RegularExpression(@"^(03|70|71|76|78|79)-\d{6}$", ErrorMessage = "Invalid Lebanese phone number format.")]

References

- Course
- Homework2.
- Homework3.
- Homework4.
- Homework5.
- Microsoft Documentation.
- Stack Overflow.