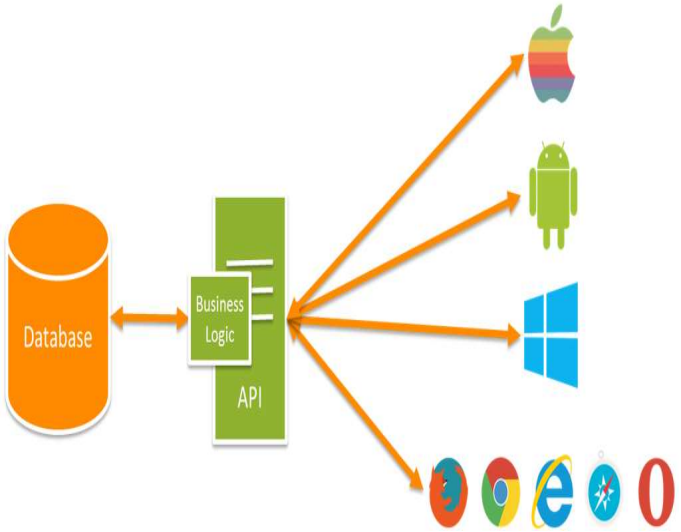
ASP.NET Core



Web API

Mohamed ELshafei

Prerequisite - Dependency injection (DI)

- **Dependency injection (DI)** is a technique for achieving loose coupling between objects and their collaborators, or dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs in order to perform its actions are provided to the class in some fashion.
- Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. This approach is known as "constructor injection"

Prerequisite - Dependency Inversion Principle (DIP)

- **The Dependency Inversion Principle (DIP)** states that high level modules should not depend on low level modules; both should depend on abstractions (typically interfaces). Abstractions should not depend on details.
- Details should depend upon abstractions which are provided to them when the class is constructed.

Prerequisite - containers

- **A container** is essentially a factory that's responsible for providing instances of types that are requested from it.

- When a system is designed to use DI, with many classes requesting their dependencies via their constructor (or properties), it's helpful to have a class dedicated to creating these classes with their associated dependencies. These classes are referred to as *containers*, or more specifically, ***Inversion of Control (IoC)*** containers or Dependency Injection (DI) containers.

Controller

- ASP.NET Core supports creating RESTful services, also known as ***web APIs***, using C#. To handle requests, a web API uses controllers.
- Controllers in a web API are classes that derive from *ControllerBase*.

- Don't create a web API controller by deriving from the Controller class. Controller derives from ControllerBase and adds support for views, so it's for handling web pages, not web API requests.

ApiController attribute

The [ApiController] attribute can be applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference
- Problem details for error status codes

```
[ApiController]
public class MyControllerBase : ControllerBase
{
}
```

Attributes

The Microsoft.AspNetCore.Mvc namespace provides attributes that can be used to configure the behavior of web API controllers and action methods.

[Route]: Specifies URL pattern for a controller or action.

[Bind] :Specifies prefix and properties to include for model binding.

[HttpGet]: Identifies an action that supports the HTTP GET action verb.

[Consumes] :Specifies data types that an action accepts.

[Produces]:Specifies data types that an action returns.

Binding source parameter inference

A binding source attribute defines the location at which an action parameter's value is found. The following binding source attributes exist:

[FromBody]: Request body

[FromForm]: Form data in the request body

[FromHeader]: Request header

[FromQuery]: Request query string parameter

[FromRoute]: Route data from the current request

Routing to controller actions in ASP.NET Core

- Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed.

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

- With attribute routing the controller name and action names play **no** role in which action is selected.

```
[Route("Home/About")]
public IActionResult MyAbout()
{
    return View("About");
}
```

Attribute routing with Http[Verb] attributes.

- Attribute routing can also make use of the *Http[Verb]* attributes such as *HttpPostAttribute*. All of these attributes can accept a route template.

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
    // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
    // ...
}
```

Token replacement in route templates

- For convenience, attribute routes support token replacement by enclosing a token in square-braces ([,]).
- No route Prefix.

```
[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

Multiple Routes

```
[Route("[controller]")]
public class ProductsController : Controller
{
   [Route("")]     // Matches 'Products'
   [Route("Index")] // Matches 'Products/Index'
   public IActionResult Index()
}
```

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
   [HttpPost("Buy")]     // Matches 'Products/Buy' and 'Store/Buy'
   [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'
   public IActionResult Buy()
}
```

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
   [HttpPut("Buy")]      // Matches PUT 'api/Products/Buy'
   [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'
   public IActionResult Buy()
}
```

Specifying attribute route optional parameters, default values, and constraints

- {id:int} integar constraint.
- {id:int?} optional Parameter
- {id:int=3} Default Value

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
    // ...
}
```

Entity Framework Core

- Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.
- EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.
- You can generate a model from an existing database, hand code a model to match your database, or use EF Migrations to create a database from your model, and then evolve it as your model changes over time.

Install Entity Framework Core

- Tools ▸ NuGet Package Manager ▸ Package Manager Console
- Run `Install-Package Microsoft.EntityFrameworkCore.SqlServer`
- To enable reverse engineering from an existing database we need to install a couple of other packages:
- Run `Install-Package Microsoft.EntityFrameworkCore.Tools`
- Run `Install-Package Microsoft.EntityFrameworkCore.SqlServer.Design`
- Run the following command to create a model from the existing database:

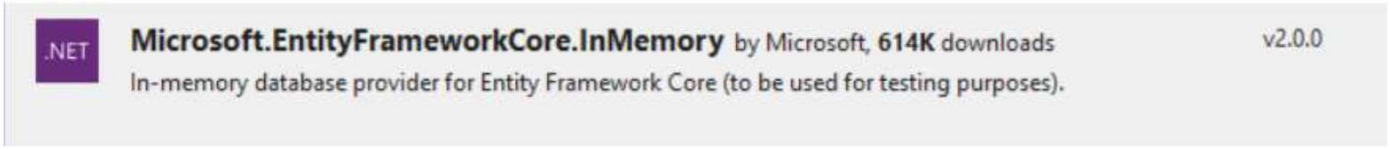
```
Scaffold-DbContext "Server=localdb;Database=Blogging;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Working With a Database Using EF Core

- Add Entity Framework Core (EF Core) in ConfigureServices:

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

- Also supports InMemory – Install this package:



and modify ConfigureServices:

```
services.AddDbContext<AppDbContext>(options =>
                options.UseInMemoryDatabase("DbName"));
```

Using EF Core’s AppDbContext(Code First)

1- install Entity Framework Core NuGet packages.

2- Create the data model

3- Create the Database Context

4-Add your connection string in appsettting.json

5- Register the context with dependency injection

6-Add Migration and Update Database.

Working With EF Core (Code First From DB)

- Tools –> NuGet Package Manager –> Package Manager Console.

- Run the following command to create a model from the existing database:

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;“
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Working with EF Core's AppDbContext

- Never directly instantiate your application's DbContext type(s)
- Instead, request from DI using constructor parameters

Recommendation

- Follow the Dependency Inversion Principle and avoid having UI types depend on *details*
  - Using Entity Framework Core is an **implementation detail**
  - Instead depend on an abstraction over your persistence method
  - Common pattern for this: **Repository**

Loading Related Data

Entity Framework Core allows you to use the navigation properties in your model to load related entities. There are three common O/RM patterns used to load related data.

- **Eager loading** means that the related data is loaded from the database as part of the initial query.
- **Explicit loading** means that the related data is explicitly loaded from the database at a later time.
- **Lazy loading** means that the related data is transparently loaded from the database when the navigation property is accessed.

Eager loading

- You can use the ***Include*** method to specify related data to be included in query results.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
    .ToList();
```

- You can include related data from multiple relationships in a single query.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
    .Include(blog => blog.Owner)
    .ToList();
```

Including multiple levels -Eager loading

- You can drill down thru relationships to include multiple levels of related data using the *ThenInclude* method.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
    .ToList();
```

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
            .ThenInclude(author => author.Photo)
    .Include(blog => blog.Owner)
        .ThenInclude(owner => owner.Photo)
    .ToList();
```

Explicit loading

- This feature was introduced in EF Core 1.1.
- When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed.
- You can explicitly load a navigation property via the DbContext.Entry(...) API.

```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

context.Entry(blog)
    .Collection(b => b.Posts)
    .Load();

context.Entry(blog)
    .Reference(b => b.Owner)
    .Load();
```

Explicit loading

```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

var postCount = context.Entry(blog)
    .Collection(b => b.Posts)
    .Query()
    .Count();
```

```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

var goodPosts = context.Entry(blog)
    .Collection(b => b.Posts)
    .Query()
    .Where(p => p.Rating > 3)
    .ToList();
```

Lazy loading

- This feature was introduced in **EF Core 2.1**.
- The simplest way to use lazy-loading is by installing the Microsoft.EntityFrameworkCore.Proxies package and enabling it with a call to *UseLazyLoadingProxies* when using AddDbContext:

```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

EF Core will then enable lazy-loading for any navigation property that can be overridden--that is, it must be *virtual* and on a class that can be inherited from.

Performance considerations

- If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved.
- In some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently.

Help Doc Using NSwag

- **NSwag**, third-party APIs that incorporate Swagger and generate a client implementation.
- **NSwag** allows you to expedite the development cycle and easily adapt to API changes.

```
Install-Package NSwag.AspNetCore
```

- **Add and configure Swagger middleware**
- register the required Swagger services

```
services.AddSwaggerDocument();
```

- enable the middleware for Swagger and Swagger UI

```
app.UseOpenApi();
app.UseSwaggerUi3();
```

http://localhost:<port>/swagger

Reference Loop Handling

- The problem was occurring because in .NET Core 3 they change little bit the JSON politics. Json.Net is not longer supported and if you want to used all Json options, you have to download this Nuget:

Microsoft.AspNetCore.Mvc.NewtonsoftJson.

```
services.AddControllers().AddNewtonsoftJson(x => x.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore);
```

Enable Cross-Origin Requests (CORS)

- Define string variable as Cors policy in ***start*** class

```
string MyAllowSpecificOrigins = "m";
```

- Register ***AddCors*** in ***ConfigureServices*** method

```
services.AddCors(options =>
{
    options.AddPolicy(MyAllowSpecificOrigins,
    builder =>
    {
        builder.AllowAnyOrigin();
    builder.AllowAnyMethod();
        builder.AllowAnyHeader();
    });
});
```

- Add ***UseCors*** Middleware in ***Configure*** method.

```
app.UseCors(MyAllowSpecificOrigins);
```

Repository Pattern In ASP.NET Core MVC And Entity Framework Core

The repository pattern is intended to create an Abstraction layer between the Data Access layer and Business Logic layer of an Application. It is a data access pattern that prompts a more loosely coupled approach to data access. We create a generic repository, which queries the data source for the data, maps the data from the data source to a business entity and persists changes in the business entity to the data source

Prerequisite - Dependency Inversion Principle (DIP)

- **The Dependency Inversion Principle (DIP)** states that high level modules should not depend on low level modules; both should depend on abstractions (typically interfaces). Abstractions should not depend on details.
- Details should depend upon abstractions which are provided to them when the class is constructed.

WHY?

- We implement repository pattern to develop a loosely coupled application. It makes the code more testable. It creates an abstraction layer between ORM and business logic layer of the application.
- the repository mediates between the data source layer (Entity Framework) and the business layer (Controller) of the application. It performs operations as following way.
  - It queries the underlying data source (database) for the data.
  - It maps the data from the data source to a business entity that uses to create the database.
  - It persists changes in the business entity to the data source.

Advantages of Repository Pattern

It has some advantages which are as follows:

- An entity might have many operations which perform from many locations in the application, so we write logic for common operations in the repository. These operations might be Create, Read, Update, Delete, Search, Filter, Sort, Paging, and Caching etc.
- The Entity Framework is not testable out of the box. We have to create mock classes to test it. Data access logic can be tested using repository pattern.
- As business logic and data access logic separate in the repository pattern that’s why it easy to manage and readable. It reduces development as well me, as common functionality logic writes once in the application.

How? “Employee Entity”

- ASP.NET Core is designed to support dependency injection. So, we create a repository interface named IEmployeeRepository for Employee .
- This interface has definitions of all methods to perform CRUD operations on the Employee entity.
- let’s implement the preceding interface on a class named EmployeeRepository.
- This class has implemention of all methods to perform CRUD operations on the Employee entity. It has a parameterized constructor that accepts Context as a parameter. It passes at a time of repository instance creates by dependency injection.

How? “Employee Entity” cont..

- web application communicates to data access layer via an interface so we register repository to the dependency injection during the application start up.
- **Create Application User Interface**
- create a view model named EmployeeViewModel for application UI and CRUD operations. This model strongly binds with a view.