Angular : client side rendering (Server reponde with data that needs to be rendered in the client side)

MVC : Server Side Rendering (Server respond with HTML pages)

Patterns:

MVC : Architectural Pattern

Contoller : receive requests then it creates Model and view then inject Model in the view

View : Depends on the Model

Model: Object

SOLID

Dependency Inversion (DI):

Higher modules shouldn't depend on lower modules, both should depend on abstraction(Abstract Class, Interface)

Inversion Of Control (IOC): Class shouldn't create an object of other class

Dependency Injection: Is a pattern that could achieve both principles (DI & IOC)

MVC Project Structure

WWWroot: static files (images/ css files/ js files/ jquery files)

Views:

Is a folder contain some razor files (extention .cshtml) these files will be rendered to html and sent to the browser when we run the program.

Naming Convention:

If we have a controller called BookController

It's better to create a folder called Book inside Views folder Views/Book

This Book folder will contain all the views related to BookController

Shared:

This folder has some files that start with _ (these files run fisrt when we run the program).

_ViewStart: is a file that defines which view will run first(by default it has the layout)

_layout: is a file that has the default layout of the program, it has a section called

@RenderBody at which the views we create will be rendered

_ViewImports: anything imported at this file will be visible to all views

Razor View

In each razor view we have to define which model will be used throughout this view

@model Book; //Book model will be used in this view

How to write C# in razor view

@Model

@() //for long expressions (example : equations)

@{} //code block in which we can define variables or functions and so on

HTML Tag Helpers

To work we must add `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers in _ViewImport`

```
Asp-action : is added to anchor tag to redirect you t a specific endpoint
<a asp-action="Add">Add a new item</a> is equivalent to <a href="book/Add">Add a new item</a>
```

If we want to send data to the view but this data doesn't exist in the view model

So we have two options to send this data:

- viewBag: is a shared dynamic object which you can access in contoller or view
- viewData: is a collection, its key is a string and the value is an object

In ASP.NET MVC, both `ViewBag` and `ViewData` are mechanisms to pass data from a controller to a view. They serve the same purpose, but they have some differences in their implementation.

1. `ViewBag`:

- `ViewBag` is a dynamic property of the `Controller` base class.

- It uses the dynamic typing feature of C# to store data.

- It uses the property syntax to store and retrieve data.

- It relies on the `DynamicObject` class to store and retrieve data at runtime.

- Since it uses dynamic typing, there is no compile-time type safety, and you need to be cautious with the data types when working with `ViewBag`.

- Example usage in a controller: `ViewBag.Message = "Hello, ViewBag!";`

- Example usage in a view: `@ViewBag.Message`

2. `ViewData`:

- `ViewData` is a dictionary-like object of the `ViewDataDictionary` class.
- It uses a key-value pair approach to store data.
- It is of type `ViewDataDictionary`, which is derived from the `Dictionary<string, object>` class.
- It uses the dictionary syntax to store and retrieve data.
- Since it uses a dictionary, you can use strongly typed values and benefit from compile-time type safety.
- Example usage in a controller: `ViewData["Message"] = "Hello, ViewData!";`
- Example usage in a view: `@ViewData["Message"]`

Both `ViewBag` and `ViewData` allow you to pass data from a controller to a view, but `ViewBag` provides a more dynamic and concise syntax, while `ViewData` provides stronger type checking and allows you to use strongly typed values. However, it's important to note that excessive use of `ViewBag` or `ViewData` is generally discouraged in favor of using strongly typed view models, which provide better code organization, type safety, and maintainability in larger projects.

SideNote: To use selectListItems in other projects (not MVC) such as in BL or DAL

We have to install **Microsoft.AspNetCore.Mvc.ViewFeatures**

Context

To use Entity Framework we have to install 3 packages:

Install-package Microsoft.EntityFrameworkCore

Install-package Microsoft.EntityFrameworkCore.SqlServer //SqlServer is the database Provider

Install-package Microsoft.EntityFrameworkCore.Tools //this package is used to allow migration

……..

When we create a context class it must inherite from DBContext

Then we create DbSet properties to representing the corresponding database tables.

There are several methods of defining DbSet properties

//Notice DbSet is a reference type so by default it allows Null

public DbSet<Product> Products { get; set; } //not recommended because you won't set the value of Products property but it will be filled from database by the corresponding table via DbContext

public DbSet<Product> Products { get; } //not recommended because it will give you a warning that Products may be null

although it won't ever be null because DbContext gets its value from the corresponding table from database and even if the table was empty, DbContext will return an empty list.

public DbSet<Product> Products { get; } = Null! // this will remove the warning by informing that you're pretty sure that products won't ever be Null, but this method is not prefered too.

//Recommended

public DbSet<Product> Products { get; } = Set<Product>

public DbSet<Product> Products => Set<Product> //Shortcut

it's recommended to use Set<> method

The Set<T> method in Entity Framework is a method provided by the DbContext class that allows you to access the DbSet<T> for a specific entity type T.

And returns table data that map to class T .

DbContext(Chatgpt)

In Entity Framework (EF), DbContext is a class that represents a session with the database, allowing you to interact with the database and perform various operations, such as querying, inserting, updating, and deleting data.

DbContext is a part of the EF Core API and is responsible for the following tasks:

1. Entity Mapping: DbContext defines the mapping between your .NET entity classes and database tables. It establishes a connection between your application's object model and the underlying database schema.

2. Change Tracking: DbContext tracks changes made to entities, enabling you to insert, update, and delete records. It keeps track of entity state changes and generates the necessary SQL statements to persist those changes to the database.

3. Querying: DbContext provides methods to query the database and retrieve data. You can use LINQ (Language Integrated Query) or query syntax to express your queries against the DbSet properties defined within the DbContext.

4. Database Operations: DbContext allows you to perform database operations such as creating, migrating, and deleting the database. It provides methods for creating the database schema, applying migrations, and executing raw SQL commands.

To use DbContext, you typically create a derived class that inherits from DbContext and represents your application's database context. This derived class includes properties that represent your entity sets (DbSet), which in turn represent database tables or views.

Here's an example of a DbContext class:

```
public class YourDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
    public DbSet<Order> Orders { get; set; }

    // Other DbSet properties and configuration
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        // Configure the database provider and connection string
        optionsBuilder.UseSqlServer("YourConnectionString");
    }
    // Other configuration methods and overrides
}
```

In the above example, `YourDbContext` is derived from `DbContext` and includes two DbSet properties: `Products` and `Orders`, representing the corresponding database tables. The `OnConfiguring` method is overridden to specify the database provider and connection string.

By creating and working with a derived DbContext class, you can interact with the database, query data, perform CRUD operations, and manage the schema using Entity Framework.

DbSet<> (Chatgpt)

In Entity Framework (EF), DbSet is a class representing a collection of entities of a specific type in the database. It is a property within a DbContext class and provides a set of methods for querying, inserting, updating, and deleting entities of that type.

DbSet is part of the EF Core API and acts as a bridge between your application's entity classes and the corresponding database tables or views. Each DbSet represents a table or view in the database and provides an interface to interact with the data stored within that table or view.

Here's an example of a DbSet property within a DbContext class:

```
public class YourDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
    public DbSet<Order> Orders { get; set; }
    // Other DbSet properties and configuration
}
```

In the above example, `YourDbContext` class includes two DbSet properties: `Products` and `Orders`. These properties represent the corresponding database tables or views for the `Product` and `Order` entities.

Once you have defined the DbSet properties within your DbContext, you can use them to perform various database operations. For example, you can query data using LINQ queries, insert new entities, update existing entities, or delete entities.

Here are some common operations you can perform using DbSet:

- Querying data:

```
var products = context.Products.Where(p => p.Category == "Electronics").ToList();
```

- Inserting a new entity:

```
var newProduct = new Product { Name = "New Product", Price = 99.99 };
context.Products.Add(newProduct);
context.SaveChanges();
```

- Updating an existing entity:

```
var product = context.Products.Find(productId);
if (product != null)
{
    product.Price = 149.99;
    context.SaveChanges();
}
```

- Deleting an entity:

```
var product = context.Products.Find(productId);
if (product != null)
{
    context.Products.Remove(product);
    context.SaveChanges();
}
```

By using DbSet properties within your DbContext, you can interact with the corresponding database tables or views, manipulate the data, and perform CRUD operations using Entity Framework.

Configuring Services

Basically we configure services in **program.cs**

If we have a conetxt like:

```
public class MyDbContext : DbContext

{

   public DbSet<Product> => Set<Ticket>();

}
```

And we want to configure it in program.cs we write

```
builder.Services.AddDbContext<MySbContext>();
but we have to add some options that provide some information such as which provider are we using
(sqlServer or MySql etc)
And what is the connection string

var connectionString = "Server=.; Database=TicketsDb; trusted_Connection=true; Encrypt=false;";
builder.Services.AddDbContext<MyDbContext>(options
    => options.UseSqlServer(connectionString));
```

```
Since we added some options to the constructor which the IOC(Inversion Of Control) Container will
create, we have to override the constructor that accepts options explicitly in the class

public class MyDbContext : DbContext

{

public MyDbContext(DbContextOptions<MyDbContext> options) : base(options)
    {

    }

    public DbSet<Product> => Set<Ticket>();

}
```

We configured our context as a service in program.cs so that if we want to use it in any class

We won't have to create object of context class (which violates the last solid principle [Dependency Inversion])

```
public class ProductsRepo : IProductsRepo
{
    private readonly MyDbContext _context;

    public DevelopersRepo()
    {
        _context = new IssuesContext();

    }

}
```

Instead we will add the context directly in the constructor and the Asp IOC container will create it with the configuration we defined in program.cs and give it to us.

```
public class ProductsRepo : IProductsRepo
```

```
{
    private readonly MyDbContext _context;

    public ProductsRepo(MyDbContext context)
    {
        _context = context;
    }

}
```

If we want to access products field we can use

_context.products

Or _context.Set<Product>()

Tracking in EF

In Entity Framework (EF), tracking and no-tracking refer to the behavior of how entities are managed and tracked by the DbContext.

1. Tracking:

- By default, EF uses tracking, which means that entities retrieved from the database are tracked by the DbContext. Any changes made to the entities are automatically detected and persisted back to the database when `SaveChanges()` is called.

- When an entity is tracked, EF keeps a reference to it and monitors any modifications made to its properties. This allows EF to generate appropriate SQL statements to update the database during `SaveChanges()`.

- Tracking allows you to work with entities in a more intuitive way, as you can modify properties directly, and the changes will be reflected in the database automatically.

2. No-Tracking:

- No-tracking means that entities retrieved from the database are not tracked or monitored by the DbContext. EF does not keep a reference to them, and any modifications made to the entities will not be automatically persisted to the database.

- When entities are retrieved with no-tracking, EF fetches the data but does not keep track of their state. This can improve performance and reduce memory usage, especially when dealing with large result sets or read-only operations.

- No-tracking is useful in scenarios where you only need to read data or when you want to prevent unintentional updates to entities.

To explicitly specify whether tracking should be enabled or disabled when querying entities, you can use the `AsTracking()` or `AsNoTracking()` methods in your queries.

Choosing between tracking and no-tracking depends on the scenario. If you need to modify and persist changes to entities, tracking is typically suitable. However, if you only need read-only access or want to improve performance, using no-tracking can be beneficial.

Service Registeration (chatgpt) + notes

In Entity Framework (EF), there are several methods provided by the dependency injection container (usually in the `IServiceCollection` interface) to register services related to EF. These methods are typically used in the configuration of the application's dependency injection container.

```
services.AddType<IYourService, YourService>();
```

//this methods states that when IyourService interface is injected into a constructor then the implementation of this interface will be (YourService).

Here are some common methods for registering services in EF:

1. `AddDbContext<TContext>`: This method is used to register the `DbContext` implementation (`TContext`) with the dependency injection container. It also allows you to configure the DbContext options. For example:

```
services.AddDbContext<YourDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("YourConnectionString")));
```

2. `AddScoped`(commonly used): This method registers a service with a scoped lifetime. When a service is scoped, a single instance is created and shared within a single scope. The scope typically corresponds to the duration of a single web request. For example:

```
services.AddScoped<IYourService, YourService>();
```

//note : AddScoped: gives only one object to all constructors which need access to this service per each http request

Ex. if controller constructor and manager constructor need access to RepoService during the same http request

both of them will be given the same object of the injected RepoService.

3. `AddTransient`: This method registers a service with a transient lifetime. Each time the service is requested, a new instance is created. Transient services are typically suitable for lightweight and stateless components. For example:

```
services.AddTransient<IYourService, YourService>();
```

//note : AddTransient: gives an object to each constructor which needs access to this service per each http request

Ex. if controller constructor and manager constructor need access to RepoService during the same http request

Each one of them will be given a different object of the injected RepoService.

4. `AddSingleton`: This method registers a service with a singleton lifetime. A single instance of the service is created and shared throughout the application. For example:

```
services.AddSingleton<IYourService, YourService>();
```

//note : AddSinglton: gives the same object to each constructor which needs access to this service for all http requests.

Ex. if controller constructor and manager constructor need access to RepoService during the different http request

Each one of them will be given the same object of the injected RepoService for all http requests

These methods can be used to register various services related to EF, such as repositories, data services, or custom components, into the dependency injection container. The choice of which method to use depends on the desired lifetime and behavior of the service being registered.

It's worth noting that the above methods are part of the default dependency injection container in ASP.NET Core, but other dependency injection frameworks may have similar or alternative methods for service registration.

Validation

Serverside validation

- You can apply server validation on View Models using data annotation (check createStudentVM on day6)
- And we can apply custom validation (API day1).

Clientside validation

- We can apply clientside form validation by adding section scripts in razor view (check Views/Students/Create.cshtml)

```
@section Scripts{
    <script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
    <script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.min.js"></script>
    }
```

- If we want to return validation message to the user before submitting the form

Ex. we want to inform the user if the email address he's trying to register with already exists in the database

We can apply remote validation on View Model

```
[Required]
    [EmailAddress]
    [Remote(action: "ValidateMail", controller: "Students")]
    public string Email { get; init; } = string.Empty;
```

```
Then we add ValidateMail endpoint in the controller(Students for example)
```

```
  public IActionResult ValidateMail(string email)
    {
        if (mails.Contains(email))
        {
            return Json($"{email} is taken");
        }
        return Json(true);
    }
```

Configuration

Images

WWWroot

Is the default provider for static files in MVC

Itworks via the middleware app.useStaticFiles();