Lesson 1

* ASP.Net .Net-in bir framework-udur. Veb layihələri, veb servisləri yaza bilərik. ASP açılışı Active Server Page-dir. Microsoft tərəfindən yaradılıb.
* .Net framework ancaq Windows əməliyyat sistemlərində istifadə olunur.
* .Net Core is Cross-platformdur.
* Veb API lar yazmaq mümkündür.
* Connected Services - hər hansı cloud qoşduqda orada göstərir.
* Dependencies – Packagelərimiz dependencylərimiz orada yerləşir.
* launchSettings.json – proyektin ilk başlaması ilə bağlı məlumatlar orada yerləşir. Hansı port ilə başlayacaq və s.
* Microsoftun işlətdiyi İnternet İnformation Service serverini işlədir (IIS Express)
* Vvv.root papkası statik faylları saxlayır. Proyekt daxilində kontenti dəyişilməyən elementlər orada yerləşir. Iconlar css fayllar static fayllar js və s.
* MVC pattern üzərindən yaradılıb. Model Vievv Controller
* Cshtml Razor Vievv-dur. Yəni HTML daxiliində CSharp kodu yazmaq mümkündür.
* @RenderBody() dəyişən componentləri orada yerləşdirir.
* Appsettings.json application ilə bağlı configuration-ları orada saxlamaq mümkündür.
* Appsettings.json configuration fayllarını hər hansı kodda dəyişiklik etmədən sadəcə fayl üzərindən dəyişmək mümkündür.
* Program.cs və Startup.cs .Net 6 dan sonra bir faylda yerləşir.
* Hər hansı servisləri qeydiyyatdan keçirmək üçün ConfigureServices hissəsindən həyata keçiririk.
* Configure isə middlevvare-ləri qeydiyyatdan keçirir.
* App.IsDeveloper yoxlayır ki proyektin başlama environmenti Developmentdir ya yox.
* App.UseDeveloperExceptionPage() xətaları göstərən middlevvare-dir.
* App.UseHttpsRedirection() bir səhifədən başqa səhifəyə göndərə bilmək mümkünlüyü.
* App.useStatic() vvv.root folderini tanıyır.
* App.useRouting() hər hansı səhifədən başqa səhifəyə route etmək mümkünlüyü.
* App.UseEndpoints()
* /home/index/{id?} burda id? Routing params sayılır. Home controller index actiondur
* Action Methodların access modifier-i public olmalıdır.
* Reflection CSharp-da olan bir texnologiyadır ki runtime vaxtı hər hansı bir classın içindən methodu tapıb onları start edə bilir. Hər bir classın meta-datası olur və o metadatanın içindən propertyləri action methodları və s. Tapa bilir.
* ASP.Net Core sırf Reflectionun üzərindən işləyir.
* ASP Requestləri pipeline üzərindən keçib sonra response qaytarır. Pipeline daxilində Middlevvarelər olur. Middlevvarelərdən yoxlamalar aparmaq üçün istifadə etmək mümkündür.
* Pipeline qayıdanda da yoxlayır çünki ola bilsin ki qayıdan vaxtı token-in vaxtı bitər və unauthorized olar.
* Custom Middlevvare-lər yazmaq mümkündür.
* Middlevvare-lər özündə RequestDelegate Next saxlıyır.
* Middlevvare-lər bir birinə chain of responsibilities pattern ilə bağlanıblar.
* Bir element digər elementin referansını saxlamaq deməkdir ChainOfResponsibility Pattern.
* Status kodlar İnformational (100-199) Success (200-299) Redirection (300-399) Client Error (400-499) Server Error (500-599)
* Əgər Action methodunun üzərində heçnə yazılmayıbsa default olaraq HttpGet olur
* HttpGetdən başqa hamısında mütləq qeyd etmək lazımdır ki method nə tipindədir.
* Put ilə Patch eyni şeydir. Patch sadəcə obyektin bir hissəsini dəyişmək üçün istifadə olunur.
* Head var o da headerdən hər hansı məlumatı çəkmək üçün istifadə olunur.
* Ok(), BadRequest(), NotFound() action resultu var.
* Vievv-a məlumat göndərəndə o nə tipdədirsə atmodel ilə istifadə edib onun tipini qeyd etməliyik vievv-da
* Məlumatın içindəki datalara çatmaq üçün atModel yazmaq lazımdır.
* Birdən çox məlumat göndərəndə VievvModel yaratmaq lazımdır.
* Tək parametrləri RouteParams ilə 2 və daha çox queryparams. İri həcmli obyekt olanda Post request ilə.
* Id? Routeparams olanda id ola da bilər olmaya da bilər.

Lesson 2

* Redirectlər zamanı yeni controllerin obyekti yaranır.
* Redirect(“/home/employees”) yönləndirmək üçün.
* RedirectToAction(“ActionName”) hansı kontroller daxilindədirsə onun içindən actionu axtarır.
* Parametr göndərmək üçün new { id = 1 } kimi obyekt göndərməliyik. Məsələn RedirectToAction(“Employee”, new { id = 1 })
* Var routeValue = nevv RouteValueDictionary(**nevv ( action = “Employee”, controller = “Home”, id = 1**) bu formada return RedirectToRoute(routeValue) etmək mümkündür
* Taghelpers vievvİmports.cshtmldə %addTagHelper ilə import olunur.
* Label-ə asp-for tag yazanda avtomatık property-name-inə bağlanmış olur
* Input içində yazılan asp-for binding edir.
* Formu post etməsi üçün buttonun type-i submit olmalıdır.
* Form-da asp-controller və asp-action tag helperləri ilə yönləndirə bilərik.
* Form method by default postdur.
* Asp tag helperlərə route params yazmaq olmur ancaq Query params yazılır.
* Validation üçün Data Annotationlardan istifadə olunur.
* Asp-validation-for ilə validationları yoxlamağa lazım olan tag helperdir. Controllerdə is Model.İsValid propertydən istifadə olunmalıdır.
* Asp-route- hissəsində nə yazılıbsa action methodunda arqument də eyni adda olmalıdır.
* TempData ilə session müddətində məlumat göndərmək mümkündür.
* VievvStart-da Vievvs papkasında olan bütün Vievvlara ortaq olan kodları yazmaq mümkündür.
* Custom Tag Helper : **TagHelper** classından törənir.
* Custom Tagın adını bildirmək üçün **[HtmlTargetElement(“employee-list”)]** kimi data annotation ilə qeyd etmək lazımdır.
* **Process** adlı method override olunur və **output** adlı bir argument var.
* **Output.TagName** ilə real hansı taga bənzəyəcək onu yazırıq. Həmin tagın daxilinə dolduracaq.
* TagHelperi göstərmək üçün **addTagHelper \*, ProjectName** ilə həmin proyektin daxilindəki bütün taghelperləri tapıb gətirəcək.

**Asp-validation-for, asp-for, asp-controller, asp-action** və s. Var.

* Custom Taghelperdə parameter göndərmək mümkündür məsələn sort=”a-z”
* Bunun üçün Sort adında prop yaradırıq. Sonra ona DataAnnotation kimi **HtmlAttributeName** dən istifadə edib adını qeyd edirik.
* .Take() LINQ ilə arraydan lazım olan sayda data götürmək mümkündür.

ViewBag

* **Type**: Dynamic property (part of the ControllerBase class).
* **Purpose**: Used to pass data from a controller to a view.
* **Scope**: Limited to the current request.
* **Syntax**:
* ViewBag.Message = "Hello, World!";

Characteristics:

* Uses dynamic type, so no need to explicitly cast types.
* Data is loosely typed (type safety isn't enforced).
* Ideal for small, temporary pieces of data.

### **ViewData**

* **Type**: Dictionary-based (from ViewDataDictionary).
* **Purpose**: Used to pass data from a controller to a view.
* **Scope**: Limited to the current request.
* **Syntax**:
* ViewData["Message"] = "Hello, World!";

**Characteristics**:

* Data is stored as key-value pairs.
* **Requires type casting when retrieving data:**
* string message = ViewData["Message"] as string;
* Better suited for structured or strongly-typed data compared to ViewBag.

### **TempData**

* **Type**: Dictionary-based (from TempDataDictionary).
* **Purpose**: Used to pass data between different requests (e.g., between a controller action and a redirect).
* **Scope**: Persists for the duration of the next request only (after that, it's cleared).
* **Syntax**:
* TempData["Message"] = "Hello, World!";

**Characteristics**:

* Useful for scenarios like redirects (where ViewBag and ViewData are cleared).
* Data is automatically removed after it's read once (unless explicitly kept using TempData.Keep() or TempData.Peek()):
* TempData.Keep("Message"); TempData.Peek("Message");
* Backed by providers (e.g., session state or cookies, depending on configuration).

### **Key Differences**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **ViewBag** | **ViewData** | **TempData** |
| **Type** | Dynamic property | Dictionary-based | Dictionary-based |
| **Scope** | Current request | Current request | Next request |
| **Type Safety** | No (dynamic) | No (requires casting) | No (requires casting) |
| **Usage** | Passing small data to views | Passing small data to views | Passing data between requests |
| **Persistence** | Not persisted after view | Not persisted after view | Persists for the next request |
| **Use Case** | Simple, dynamic data | Structured data | Data for redirects, post-redirect-get (PRG) |

### **When to Use Each**

1. **Use ViewBag** when:

* You need quick, dynamic, and lightweight data transfer to a view.
* Type safety isn’t a concern.

1. **Use ViewData** when:

* You prefer key-value pairs for passing data.
* You want slightly more structure and don't mind type casting.

1. **Use TempData** when:

* You need to pass data across requests (e.g., during redirects).
* You want to store temporary data that should expire after being read.

### **TempData Scope Explanation**

* TempData is designed to **persist data only for the next request**.
* The data in TempData is removed once it is **read**. If not read, it will persist only for the subsequent request.
* However, you can explicitly keep the data using TempData.Keep() or TempData.Peek().

### **How TempData Works**

1. **When TempData is Read**:

* Once you access a TempData value in a controller or a view, it is **marked for deletion** at the end of the request.
* On subsequent requests, that value will no longer be available unless you explicitly call TempData.Keep().

1. **When TempData is Not Read**:

* If you don’t access the TempData value, it persists to the next request.

1. **How it is Stored**:

* TempData is typically stored using a provider, like **Session State** or **Cookies**, depending on configuration.

### **Examples**

#### **Example 1: Basic TempData Behavior**

**Controller Code:**

csharp

CopyEdit

public IActionResult ActionOne() { TempData["Message"] = "This is TempData!"; return RedirectToAction("ActionTwo"); } public IActionResult ActionTwo() { string message = TempData["Message"] as string; // Reads and deletes "Message" return View(); }

**Behavior**:

* When ActionOne redirects to ActionTwo, TempData["Message"] is available in ActionTwo because it is the next request.
* Once TempData["Message"] is accessed in ActionTwo, it is **deleted**, and it will no longer be available for further requests.

#### **Example 2: TempData Not Accessed**

**Controller Code:**

csharp

CopyEdit

public IActionResult ActionOne() { TempData["Message"] = "This is TempData!"; return RedirectToAction("ActionTwo"); } public IActionResult ActionTwo() { // TempData["Message"] is not accessed here. return RedirectToAction("ActionThree"); } public IActionResult ActionThree() { string message = TempData["Message"] as string; // Still available because it wasn't accessed in ActionTwo. return View(); }

**Behavior**:

* TempData["Message"] will persist through the redirect from ActionTwo to ActionThree because it was **not read** in ActionTwo.

#### **Example 3: Using TempData.Keep()**

If you want to access TempData but still keep it for future requests, you can use TempData.Keep().

**Controller Code:**

public IActionResult ActionOne() { TempData["Message"] = "This is TempData!"; return RedirectToAction("ActionTwo"); }

public IActionResult ActionTwo() { string message = TempData["Message"] as string; // Accesses "Message" TempData.Keep("Message"); // Keeps "Message" for the next request return RedirectToAction("ActionThree"); }

public IActionResult ActionThree() { string message = TempData["Message"] as string; // "Message" is still available return View(); }

**Behavior**:

* The TempData["Message"] value remains available even after it is accessed in ActionTwo, thanks to TempData.Keep().

#### **Example 4: Using TempData.Peek()**

If you want to read a TempData value without marking it for deletion, you can use TempData.Peek().

**Controller Code:**

csharp

CopyEdit

public IActionResult ActionOne() { TempData["Message"] = "This is TempData!"; return RedirectToAction("ActionTwo"); }

public IActionResult ActionTwo() { string message = TempData.Peek("Message") as string; // Reads without marking for deletion return RedirectToAction("ActionThree"); }

public IActionResult ActionThree() { string message = TempData["Message"] as string; // Still available because it was Peeked, not read return View(); }

**Behavior**:

* The Peek() method allows you to check the value of TempData without removing it, so it remains available for subsequent requests.

### **When TempData Gets Deleted**

1. **Once Read**:

* TempData is deleted as soon as its value is accessed (unless Keep() or Peek() is used).

1. **After the Next Request**:

* If TempData is not accessed at all in the next request, it will still be cleared afterward.

1. **Session Expiration**:

* If TempData is backed by session state, it will be lost if the session expires.

1. **When Stored in Cookies**:

* If using cookie-based storage, it will persist until the cookie expires.

### **Practical Scenarios for TempData**

* **Flash Messages**:
* For showing one-time messages like success/error notifications after a redirect.
* csharp
* CopyEdit
* TempData["Success"] = "Profile updated successfully!"; return RedirectToAction("Profile");
* **Post-Redirect-Get Pattern**:
* TempData is perfect for preserving data during a RedirectToAction in the PRG pattern.

### **Summary**

* **TempData persists for the next request only**, and it is cleared after being accessed.
* Use TempData.Keep() or TempData.Peek() to prevent automatic deletion.
* It is ideal for scenarios where data must persist across a redirect or for one-time operations.

Lesson 3

* SOLID-in 5 prinsipi var  
  Single Responsibillity  
  Open-Close principle  
  Liscov Substition  
  Interface Aggregation  
  Dependency Inversion
* Dependency Inversion (Dependency Injection) – asılılığı konkret tiplərdən yox abstract tiplərə çevirir.
* Hər inject olunan obyektın yaşam müddəti var.
* SOLID prinsipləri obyekt yönümlü proqramlaşdırmaya əsasən yaradılmış prinsiplərdir.
* Yaşam müddətlərinin **3 növü** var. Generic olaraq interface-i və classı göstərmək lazımdır  
  **Add Singleton  
  Add Scope  
  Add Transient**
* Services papkasında CRUD əməliyyatından başqa nə əməliyyatlar varsa orada yerləşir.
* CRUD əməliyyatları Repository-də yazılır.
* **AddSingleton** yalnız sistemdə bir dəfə yaranır və hamı bir ortaq obyekti istifadə edir.
* **AddScoped** hər sorğu üçün ayrıca bir dənə obyekt yaradır və Request dövründə əmin obyektdən istifadə edir.
* **AddTransient** hər sorğu üçün ayrıca bir dənə obyekt yaradır və həmçinin request daxilində başqa yerdə də inject etsə yenə də yeni bir obyekt yaradacaq.
* **SelectListItem** ASP.Net Core-da **Combobox**-un yaranmasına kömək edən bir tipdir. Text və Value propertyləri var.
* **Select tagında** **asp-for** yazıb **CityId** göstəririk ki seçilmiş value **CityId-yə bind** olunsun. **Asp-items** isə **List SelectedListItem** ı yazıb **bind** etmək lazımdır.
* **Builder.Build()** **hissəsinə qədər servislərin qeydiyyatı** olan hissədi builderdən sonra isə middlevarelərin qeyd olunduğu yerdi.
* **Builder.Configuration ilə appsettings.json** üzərindən məlumatlar almaq mümkündür.
* **Builder.Services.AddDbContext** ilə DbContextə servis kimi əlavə edirik.

1. **Thread.Sleep(milliseconds)**

What it does: Suspends the execution of the current thread for a specified duration.

How it works:

The thread is completely blocked during the delay.

It does not allow other tasks to run on the same thread while it's sleeping.

Usage:

Used for short, blocking delays.

Often used in console or simple applications where blocking a thread is acceptable.

Drawbacks:

Wastes system resources by blocking the thread.

Not ideal in asynchronous or UI-based applications, as it can freeze the UI or block other tasks.

2. **Task.Delay(milliseconds)**

What it does: Creates a delay without blocking the thread.

How it works:

Internally uses a timer to track the delay.

The thread is free to perform other tasks while waiting for the delay to complete.

Works well with asynchronous programming (async/await).

Usage:

Used in asynchronous code where non-blocking behavior is required.

Preferred in modern applications for delays in UI or server-based systems.

Benefits:

Doesn't block threads, allowing for better scalability and responsiveness.

Works seamlessly with async/await for clean, non-blocking code.

Lesson 4

* Razor Page-da ancaq **pagelər var** və hər bir page-in öz **code-behind** faylı var. Yəni vievv kodunun arxasındakı məntiq orada yazılır.
* By default **OnGet() və OnPost** methodları var. OnGet methodu səhifə **çağırılan anda işə** düşür.
* Page-də **propertylər by default one vvay binding** olduğu üçün arxa planda dəyişiləndə vievv-da görsənir lakin **əksinə olanda arxa planda dəyişmir**.
* Tvvo vvay binding etmək üçün Data Annotation olaraq üstünə **BindProperty** yazmaq lazımdır.
* VievvComponents nədir? Əgər hər hansı bir modul varsa və hər hansı bir dinamik datalardan (databasedən gələn yəni dəyişən məlumatlar) istifadə edirsə, o zaman normal **controller tərəfindən hər dəfə məlumatı çağırmaq düzgün deyil**. Bunun üçün componenti çağırmaq lazımdır ki həmin componentin içində bir dəfə sorğu getsin və component layout-da görünsün.
* **VievvComponents papkası olmalıdır**.
* İstənilən component : **VievvComponent** classından törədilməlidir.
* Public **VievvVievvComponentResult InvokeAsync()** methodu olmalıdır.
* **Shared** papkasının içində **Components** papkası olmalıdır. Vievv lar orada yazılır. O papkanın altında **Componentın adına uyğun papka yaradılıb** içində **Default.cshtml** yaradılır.
* Componenti çağırmaq **@avvait Component.InvokeAsync(“ComponentNameHere”)**
* Areas onun üçün istifadə olunur ki proyekt böyük olduqda fərqli modullar ola bilər məsələn **Klient və Admin hissələri və hər birinin özünün ayrıca Vievv-ları Controllerləri \_Layoutları** və s. Olur. Area-ya görə hansını default götürəcəyi seçilir.
* **Areas** papkasında yazılır.
* Hər bir area üçün ayrıca **Controllers Vievvs** olur
* Controlleri yaradandan sonra classın başında DataAnnotation olan məsələn **[Area(“Admin”)]** yazılır
* Tanıtmaq üçün **app.MapControllerRoute** içində **name: “areas”** olur və pattern hissəsində **{area:exist}/{controller=Home}/{action=Index}/{id?}** yazılır.
* Razor pages məntiqi var. Orada controller məntiqi olmur. Hər bir pagelərin öz code behind faylı olur. Öz code behind məntiqində hansı propertyləri yazsaq hamısı cshtml faylının içində model olaraq görsənəcək.
* Default olaraq OnGet və OnPost methodları var. Səhifə çağırılanda OnGet methodu işə düşür. Səhifə daxilində hər hansı yerdə post methodu işləndikdə isə OnPost methodu işə düşür.
* Default olaraq yaratdığımız propertylər one vvay binding prinsipi ilə işləyir. Yəni Code Behind-da dəyişilən property vievv da görsənir lakin vievv da dəyişilən property code behindda dəyişmir.
* Tvvo vvay binding etmək üçün property-nin üzərində [BindProperty] yazmaq lazımdır.

Lesson 5

* Bootstrap front end tərəfini çox sürətlə inkişaf etdirməyə imkan verir.
* Bootstrap scss-də yazılıb.
* Bootstrapı ayrıca yükləmək də olar CDN üzərindən gətirmək də olar. CDN Hər hansı yerdə quraşdırılmış serverdə yerləşməsi deməkdir.
* Monolit arxitekturası – Proyekt daxilində başqa class librarylər olur. Data Access layer başqa bir proyekt olur. Business layer – servislərin yazıldığı yer başqa bir class library olacaq.
* Proyektləri bölmək bizim üçün nəyə kömək edir? Əgər biz nə vaxtsa MVC proyektindən başqa bir proyektə məsələn WPF və s. Kimi proyektə keçəndə biz sadəcə UI-ı dəyişmiş oluruq. Amma bu Monolit arxitekturasından istifadə etməsək, bütün kodları MVC-nin içində yazsaq məcbur hər şeyi silib yenidən yazmalı olacıq. Class library-lər buna görə vacibdir.
* Generic EfEntitiyRepositoryBase olduğu üçün context.Products və s. Kimi propertylərinə çata bilmirik. Bunun üçün Entry var.
* \_context.Entry<Tentity> ilə entity qeydə alınır və ardınca bu methoddan gələn entity-nin state-ni dəyişirik. Added. Updated. Deleted və s. Olduğunu deyib savechanges edirik.
* \_context.Set<Tentity>.Methods() var həmin propertylər üzərindəki methodları çağırmağa imkan verir. Generic olaraq.

Lesson 6

* Monolit arxitekturasına niyə keçdik? (Sual)
* Monolit arxitekturasında sorğu hansı layerdən hansı layerə gedir? (Sual) Birinci UI layerə gəlir sorğu, sonra Business layerə gedir, sonra Data Access layer.
* Generic Repository Pattern nədir. Nə üçün istifadə olunur? (Sual) Bir interface üzərindən bütün interfacelərə aid pattern qurulur. Bununla da təkrar hər bir interface-ə CRUD yaratmağa ehtiyac olmur.
* VievvComponent bizə nə üçün lazımdır. (Sual) Dinamik datalardan istifadə olunursa onda VievvComponent yaratmaq daha məntiqidir. (VievvComponent haqqında əvvəlki note-lərdən oxu, necə yaratmaq lazımdı hansı mərhələlərdən keçirilir onu soruşub)
* Dependency Injection nədir və nə üçün istifadə olunur? Hansı problemləri həll edir və s.

Lesson 7

* Session storage – açılan brauzerin yaddaşında saxlanılır, yəni RAM-da.
* Session-dan istifadə etmək üçün Isession adlı interface var.
* Isession interface-ində Set(Get)String və Set(Get)Int methodları var ki hansı ki string və ya int yaza bilir məlumat kimi.
* Obyekti yazmaq üçün əlavə extention method yazmalıyıq hansı ki obyekti JSON-a çevirib string kimi onu SetString ilə təyin edək.
* Sessionu aktivləşdirmək üçün builder.Services.AddSession qeyd edirik və app.UseAuthorization() dan sonra app.UseSession() methodunu çağırırıq.
* HttpContext controllerin falan içində bir başa yazmaq olur lakin başqa sadə .cs faylının içində yazmaq üçün IhttpContextAccessor adlı interface-i inject etmək lazımdır.
* Animate.style adlı saytda çoxlu animationlar var.

Lesson 8

* Session nədir? (Sual)
* Sessiona çatmaq üçün nədən istifadə edirik? (Sual) Isession interface-i var. HttpContext.Session üzərindən istifadə olunur.
* Hansi methodları var? (Sual) SetInt32 və SetString və onların Get-ləri
* Objecti yazmaq üçün nə edə bilərik? (Sual) Extention method yaza bilərik.
* HttpContext nədir? (Sual) Request baş verdiyi zaman HttpContext yaranır. HttpContext cari request ilə bağlı bütün məlumatları özündə saxlayır məsələn, Headerlər cookielər Requestin özü Requestin body-si, Response, Query-də gələn paramslar, Routes-də gələn paramslar və s.
* ASP.NET Core da Identity modulu var. O bizə sırf Authentication və Authorization proseslərində kömək edəcək.
* Authentication nədir? Useri signup və sign in olma məsələləridir. Yəni qeydiyyatdan keçirib giriş edir.
* Authorization isə artıq user giriş etdikdən sonra onun rollarıdır. Rollara görə müəyyən sahələrə limit qoyulur.
* CustomIdentityUser : **IdentityUser** classından törənir. Bu class **AspNetCore.Identity** **namespace**-indən gəlir.
* **IdentityUser** yaradılırsa **mütləq** onun **IdentityRole**-si **olmalıdır**. Mütləq hər userin rolu olur.
* CustomIdentityRole : **IdentityRole** classından törənir. Eyni namespace-dədir.
* CustomIdentityDbContext : **IdentityDbContext** classından törənir. Bu DbContexti yükləmək üçün **Microsoft.AspNetCore.Identity.EntitiyFrameworkcore** **package-i lazımdır.**
* Həmin IdentityDbContext-ə user və rolu göstərmək lazımdır bunun üçün **IdentityDbContext<CustomIdentityUser, CustomIdentityRole, string>** yaza bilərik. Bunu yazdıqdan sonra ayrıca **DbSet<CustomIdentityUser> və DbSet<CustomIdentityRole> yazmağa ehtiyac yoxdur**.
* Yuxarıdakı bənddə **string** yazmağımız onu göstərir ki həmin table-lərin unique ID si necə olacaq. **String olsa Guid** **int olsa 1,2**. Sequence verir primary key kimi.
* **Builder.Services.AddIdentity** generic olaraq göstərmək lazımdır hansı class olduğunu. Məsələn CustomIdentityUser, CustomIdentityRole. Ümumi yazılış aşağıdakı kimidir.

**Builder.Services.AddIdentity<CustomIdentityUser, CustomIdentityRole>()**

**.AddEntityFrameworkStores<CustomIdentityDbContext>()**

**.AddDefaultTokenProvider()**

* Normalda token özümüz generate edirik. Olmadığı üçün avtomatik AddDefaultTokenProvider ilə əlavə etdiririk.
* **App.UseAuthorization()** dan əvvəl **app.UseAuthentication()** istifadə edilməlidir mütləq. Çünki birinci Authenticate olunur sonra Authorize olunur.
* Add-migration zamanı fərqli contextlər olsa onda **add-migration -Context “ContextName”** yazmaq lazımdır ki bilinsin hansını migrate etməlidir.
* Hash kodlama alqoritmidir. Kodladıqdan sonra onu geri dönüşümü mümkün deyil. Buna görə həmişə user tərəfindən daxil olunmuş passvvord hash koda şifrələnib database ilə yoxlanılır. Müqayisə Hash ilə gedir.
* AccountControllerin içində qeydiyyat və loqin məsələləri aparılır.
* Sign in Sign up prosesslərini aparmaq üçün xüsusi classlar var. **UserManager** classı. User ilə bağlı prosesslərin aparılmasına kömək edir.
* **RoleManager** classı rolların idarə olunması məsələlərinə baxır.
* **SigninManager** sign in sign up prosessl
* **UserManager** generic olaraq hansı classı idarə edəcəyini istəyir. Hal-hazırki proyektdə CustomIdentityUser daxil edirik.
* **RoleManager** də həmçinin eyni qaydada.
* **IdentityResult** qaytaran bir method var. userManager.CreateAsync(CustomIdentityUser, passvvord) methodundan istifadə etməklə userManager-ə userManagerin işlətdiyi CustomIdentityUser obyektini göndərmək mümkündür. Yanında passvvord da göndəririk. Həmin CreateAsync methodu bizə IdentityResult return edir.
* **userManager**-in özünün passvvord check etməsi var. Böyük hərf kiçik hərf rəqəm və simvoldan ibarət olmalıdır şifrə.
* Sistemə yaradılan hər bir userin bir rolu olmalıdır. Ona görə əgər user yaratmaq mümkün olduqda IdentityResult olaraq onun nəticəsini götürüb yoxlamaq lazımdır. Succeed olsa davam edib rol təyin etməliyik.
* **roleManager-də** RoleExistAsync() methodu var string qəbul edir. Yoxlayır əgər X rol var ya yox. True false qaytarır.
* **roleManager.CreateAsync()** methodu ilə CustomIDentityRole yaradıb göndəririk. Bu method da IdentityResult qaytarır.
* **userManager.AddToRoleAsync(CustomIdentityUser, “RoleString”)** qəbul edir hansı ki göstərilən user-ə role təyin edir.
* **userManager.CreateAsync()** methodu nə vaxt uğursuz deyə bilər, məsələn, əgər eyni username varsa və ya passvvord pattern düzgün deyilsə və s. kimi problemlərdə uğursuz olacaq.
* **Register Post** action methoduna **[ValidateAntiForgeryToken]** data annotation yazmaq lazımdır. Bu annotation o deməkdir ki, bu form yanlız bizə aid olduğu zaman işə düşür. Yəni, əgər brauzerdən açılmayıbsa bir başa postman-dan sorğu göndərilirsə qəbul edilmir post sorğusu. ASP-də form açılan zaman bir token yaradılır, bu token həm özündə saxlayır həmdə clientə verir. Ancaq həmin token ilə post edilərkən method işə düşür. Əks halda işə düşməyəcək.
* **SigninManager** üzərindən PassvvordSignInAsync() methodu var parameter kimi username, passvvord, rememberMe(boolean), false göndərilir. Yenə result verir IdentityResult
* Əgər hər hansı controllerə bizə authorization lazımdırsa həmin controllerin üstünə **[Authorize]** yazırıq. Bu zaman controller daxilindəki bütün action methodlarına aid olacaq. Və ya xüsusi hər hansı bir action methodunun üstünə də yaza bilərik.
* Əgər [Authorize] yazmışıqsa və debug-dan sonra yenə də authorize olmadan açırsa səhifəni deməli cache-i təmizləmək lazımdır. Çünki bizim öncəki məlumatlarımız cache-də yadda qalır.
* Əgər Authorize deyilsənsə Avtomatik olaraq **Account** Controllerində Login Action methoduna yönləndirir. **Account/Login** axtarılır avtomatik.
* Authorize mötərizələrdə parameter kimi rolun adını yazmaq mümkündür. O zaman ancaq həmin rolu olan user-lər girə bilir.
* Bir neçə rol olduqda isə vergül ilə bir birindən ayrılmış şəkildə mötərizədə rolların adları qeyd olunur.
* [Authorize(Roles = “Admin”)] kimi də yaza bilərik.

Lesson 9

* Qeyd yoxdur

Lesson 10 (Web API, RESTful API)

* Web API nədir? API bir giriş qapısıdır. Hər hansı bir məlumata çatmaq üçün giriş qapısıdır.
* RESTFul API nədir? API yazmaq prinsipidir. Hansı ki orda HttpGet Post Put Delete və s. Var. RESTFul API-in vacib prinsipləri var.
* Web API -da Models və Vievvs yoxdur.
* Web API Controllerlər ControllerBase-dən törənir. Amma MVC-də Controller-dən törənirdi.
* ControllerBase-in içində vievv anlayışı yoxdur. Ancaq content qaytarır.
* Controllerin üzərində [ApiController] yazılır və [Route(“[controller]”)] yazılır. Bildirilir ki bu bir controllerdir. Api/controller kimi də yazıla bilər.
* Web API-da eyni adlı method yazmaq mümkün deyil. MVC-də məsələn Register-ə həm get həm də post yazmaq olurdusa burda mümkün deyil ki eyni ada həm get həmdə post yazasan. Methoda konkret Endpoint adı vermək lazımdır .
* HttpGet endpointin içində yazılmasa onda avtomatik Get() methodunu işə salır.
* Swagger bizim controllerlərin dokumentasiyasıdır.
* RestFul principles
  1. Statelessness – Endpointlərin bir birilə asılılığı olmamalıdır. Client-dən Server-ə gedən bir məlumat tam qəbul olunmalıdır. Yəni client nə qədər field göndərirsə server də o qədər field qəbul etməlidir. 5 göndərib 10 dənəlik class-da 5-ini saxlamaq düzgün deyil.
  2. Düzgün status code qaytarılmalıdır.
  3. **Global Error Handling** - Errorları catch etmək üçün ümümi bir middleware yazılmalıdır.
  4. Endpointlərdə bacardığımız qədər isimlərdən istifadə etmək lazımdır. Məsələn AddUser yox Users-ə HttpPost etmək lazımdır.
  5. Düzgün Http Methodlardan istifadə etmək lazımdır. Post olmalıdırsa Post yazın Deletedirsə HttpDelete et.
  6. Cache istifadə olunması. Cache istifadə etmək üçün IMemoryCache interface-I var. Cache-dən istifadə edəndə performansı qaldıracaq.
* Idempotent nədir? Bəzən elə olur ki eyni requestlər dəfələrlə təkrarlanır, məsələn 1 saniyəyə 100lərlə eyni request gəlir. Bu zaman ASP özü database-ə sorğunu bir dəfə göndərir və 100 dəfə eyni resultu response qaytarır.

Post, Patch Idempotent deyil. Get Delete Put Idempotentdir. Bu özəllik ASP.NET-in özündə var.

* [HttpGet(“{id}”)] buradakı route paramsdır.
* HttpPost-da parametrlər by default FromBody-dən götürülür.
* Əgər hər hansı bir property nullable deyilsə onu boş göndərmək olmaz. Yoxsa Error alacıq.
* FromBody FromForm FromQuery var. FromBody-Post zamanı body-də göndərilən obyektdən götürür. FromForm sırf form üzərindən göndərilən obyektdən götürür. FromQuery-də queryparamsdan gələn məlumatı götürür.
* DTO-lardan istifadə etmək lazımdır. Data Transfer Object deməkdir.