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.

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

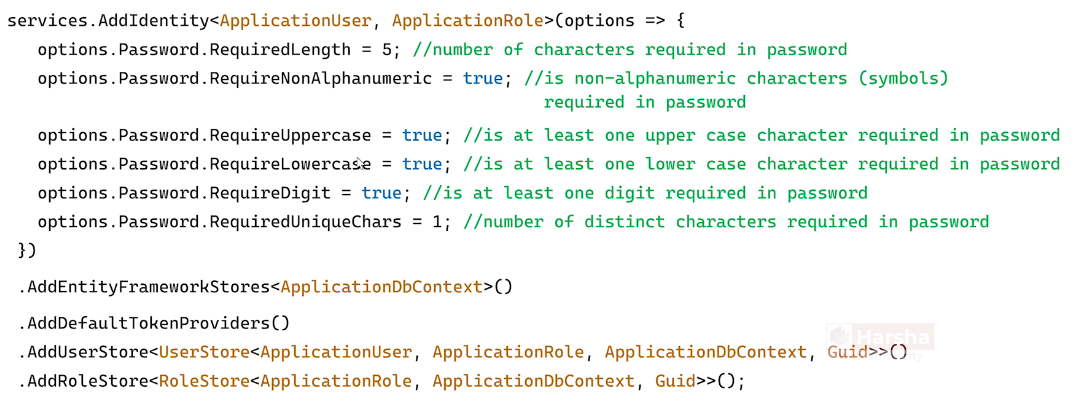
* Session nədir? (Sual)
* Sessiona çatmaq üçün nədən istifadə edirik? (Sual) Isession interface-i var. HttpContext.Session üzərindən istifadə olunur.
* Hansi methodları var? (Sual) SetInt32 və SetString və onların Get-ləri
* Objecti yazmaq üçün nə edə bilərik? (Sual) Extention method yaza bilərik.
* 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.
  7. Pagination olmalıdır mütləq.
* 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.

Lesson 11

* **Web API nədir?** Web API (Web Application Programming Interface) veb üzərindən xidmətlər təqdim edən bir interfeysdir. O, müxtəlif proqramlar və sistemlər arasında məlumat mübadiləsini təmin edir. Web API-lər adətən HTTP protokolu vasitəsilə çalışır və məlumatları JSON və ya XML formatında ötürür.
* **Restful API nədir?** RESTful API, REST (Representational State Transfer) prinsiplərinə əsaslanan bir veb API növüdür.

REST API HTTP metodları vasitəsilə məlumatları idarə edir:

HTTP Metodu Təsviri Misal (URL)

GET Məlumatı əldə edir /api/users – Bütün istifadəçiləri gətirir

POST Yeni məlumat əlavə edir /api/users – Yeni istifadəçi əlavə edir

PUT Mövcud məlumatı yeniləyir /api/users/1 – ID-si 1 olan istifadəçini yeniləyir

DELETE Məlumatı silir /api/users/1 – ID-si 1 olan istifadəçini silir

* **Hansı status kod qrupları var?**

1. 1xx - İnformasiyalı Cavablar

Bu kodlar sorğunun qəbul edildiyini və işlənməyə davam etdiyini bildirir.

Kod Açıqlama

100 Continue Server sorğunun bir hissəsini qəbul edib, davam etməyə hazırdır.

101 Switching Protocols Server yeni bir protokola keçməyi qəbul edib.

103 Early Hints Brauzerə resursların yüklənməsinə başlaması üçün ilkin məlumat verir.

2. 2xx - Uğurlu Cavablar

Bu kodlar sorğunun uğurla yerinə yetirildiyini göstərir.

Kod Açıqlama

200 OK Sorğu uğurla icra olundu və cavab qaytarıldı.

201 Created Yeni resurs yaradıldı (məs., POST sorğusu ilə).

202 Accepted Sorğu qəbul edildi, amma hələ icra olunmayıb.

204 No Content Cavab qaytarılmır, lakin sorğu uğurludur.

3. 3xx - Yönləndirmə (Redirection) Cavabları

Bu kodlar müştəriyə başqa bir URL-ə keçməli olduğunu bildirir.

Kod Açıqlama

301 Moved Permanently Resurs daimi olaraq yeni ünvana köçürülüb.

302 Found (Moved Temporarily) Resurs müvəqqəti olaraq başqa ünvandadır.

304 Not Modified Keşlənmiş versiya istifadə oluna bilər, yenisi yüklənməsin.

4. 4xx - Müştəri Xətaları (Client Errors)

Bu kodlar müştəri tərəfindən edilən səhvləri bildirir.

Kod Açıqlama

400 Bad Request Sorğu səhv və ya yanlış formatdadır.

401 Unauthorized Giriş icazəsi tələb olunur.

403 Forbidden Resursa giriş qadağandır.

404 Not Found Resurs tapılmadı.

405 Method Not Allowed HTTP metodu bu resursa icazə verilmir.

5. 5xx - Server Xətaları (Server Errors)

Bu kodlar serverin sorğunu yerinə yetirə bilmədiyini göstərir.

Kod Açıqlama

500 Internal Server Error Server daxilində gözlənilməz xəta baş verdi.

502 Bad Gateway Server başqa bir serverdən düzgün cavab ala bilmədi.

503 Service Unavailable Server müvəqqəti olaraq mövcud deyil.

504 Gateway Timeout Server, başqa bir serverdən cavab alarkən vaxtı bitdi.

Ən çox istifadə edilən status kodları

✅ 200 OK – Sorğu uğurludur.

✅ 201 Created – Yeni resurs yaradıldı.

✅ 400 Bad Request – Sorğu səhvdir.

✅ 401 Unauthorized – İcazə tələb olunur.

✅ 403 Forbidden – Giriş qadağandır.

✅ 404 Not Found – Resurs tapılmadı.

✅ 500 Internal Server Error – Serverdə gözlənilməz xəta var.s

* **DTO-lar nə üçün istifadə olunur?** DTO (Data Transfer Object), sistem daxilində və ya xarici sistemlərlə məlumat mübadiləsi zamanı istifadə olunan obyektlərdir. DTO-lar əsasən məlumatın təhlükəsiz, optimallaşdırılmış və strukturlaşdırılmış şəkildə ötürülməsi üçün istifadə olunur.
* **Idempotent nədir?**

Idempotentlik, eyni əməliyyatın bir və ya bir neçə dəfə icra olunmasının eyni nəticəni verməsi prinsipidir. Yəni, eyni sorğunu dəfələrlə göndərsən belə, nəticə dəyişməməlidir.

Idempotent olan və olmayan HTTP metodları

HTTP Metodu Idempotent? İzah

GET ✅ Bəli Eyni GET sorğusunu neçə dəfə göndərsən, nəticə dəyişmir.

PUT ✅ Bəli Mövcud resursu yeniləyir. Neçə dəfə göndərsən, nəticə eyni olacaq.

DELETE ✅ Bəli Eyni resursu silmək üçün bir neçə dəfə çağırsan da, ilk uğurlu çağırışdan sonra dəyişiklik olmayacaq.

POST ❌ Xeyr Hər çağırış yeni resurs yarada bilər, ona görə də nəticə dəyişə bilər.

PATCH ❌ Xeyr Mövcud resursun müəyyən hissəsini dəyişdirdiyi üçün nəticə fərqli ola bilər.

Lesson 11 yeni mövzuya giriş

TextOutputFormatter sinifi, ümumiyyətlə ASP.NET və ya digər proqramlaşdırma mühitlərində mətn formatında məlumatların çıxarılması və formatlanması ilə əlaqəli sinifdir. Bu sinif, verilən məlumatları JSON, XML və ya digər formatlarda təqdim etmək əvəzinə, sadə mətn formatında təqdim edir. Əsasən, HTTP cavablarını mətndə göstərmək üçün istifadə olunur.

ASP.NET MVC və ya Web API tətbiqlərində TextOutputFormatter tipli formatter'lər, serverin istifadəçiyə geri göndərdiyi məlumatların necə formatlanacağını müəyyən edir. Məsələn, bir çox tətbiqdə API cavabları JSON formatında olsa da, bəzi hallarda bu cavablar sadə mətn formatında olmalıdır. Bu halda, TextOutputFormatter istifadə edilə bilər.

* OutputFormatter : TextOutputFormatter-dən törənir.
* Konstruktorunda MediaFormat-ı və s. qeyd oluna bilər məsələn text/vcard və s.
* Bunun üçün SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse(“text/vcard”)) yazmaq lazımdır. MediaTypeHeaderValue Microsoft.Net.Http.Headers-dən gəlir.
* SupportEncodings hansı encodingləri dəstəkləyir onu qeyd etmək üçündür.
* WriteResponseBodyAsync(OutputFormatterWriteContext context, Encoding selectedEncoding) ilə əsas işi görürük.
* Var response = context.HttpContext.Response ilə responseni götürürük.
* context.Object üzərindən response-nin tipini öyrənə bilərik.
* Hər bir tip üçün ayrıca yoxlayırıq yəni context.Object is Ienumerable<StudentDto> list və ya individual yoxlaya bilərik.
* Biz OutputFormatter-imizi sistemə tanıtmalıyıq bunun üçün də Program.cs də AddControllers yerində options açırıq. Açılmış options.OutputFormatters.Insert(0, new VcardOutputFormatter()) yazaraq ilk bizim formatterimizi qeydiyyata alırıq.

Lesson 13 Custom Middleware

* Hər bir requestimiz proqramın içindəki pipeline-dan keçir. Bu pipeline içərisində middlevarelərimiz var.
* Hər bir middlevare özündən sonrakı middlevare-i next delegate olaraq yadında saxlayır.
* Middlevare-lərin məqsədi müəyyən yoxlamalar etmək və ya fərqli əməliyyatlar etmək ondan sonra növbəti next middlevare-ə keçməkdir.
* Həm requestdə həm də response-də yoxlanılır. Daha secure olması üçün.
* Custom Middlevare yazdığımız zaman mütləq içində RequestDelegate next olmalıdır və Invoke methodu olmalıdır hansı ki HttpContext qəbul edir.
* Filter middlevare-in daha yüngül variantıdır. Xüsusi hər hansı controllerə, və ya action-a aid edə bilərik.
* Middlevare-i əlavə etmək üçün app.UseMiddleware<MyMiddlewareClass>();
* Jwt token-də nə zaman ki sorğu göndərilir, Headers-də Authorization-a Bearer “token” göndərmək lazımdır.
* HttpContext.Items var Dictionary kimi key value prinsipi ilə işləyir. Token yoxlayıb Users key-inə userimizi əlavə edə bilərik əgər validdirsə. Bunun üzərindən də yoxlama aparmaq mümkündür.

Lesson 14.

* .Include() eləməklə DbContext-də Eager loading ilə reference olmuş klassları da gətirir.
* .SequenceEqual() 2 dənə arrayin içindəki elementləri dənə dənə bir birinə yoxlayır.
* AutoMapper avtomatik bir class-dan başqa class-a keçirə bilir.
* Bir dənə AutoMapper adlı class yaradıb onu Profile classından törədirik. (AutoMapper içərisindəki Profile classından)
* Constructorda isə CreateMap<FromClass, ToClass>() hansı classdan hansı classa keçəcəyini göstəririk.
* ReverseMap edəndə FromClass ToClass əksinə də qeyd edir yəni ToClassdan FromClassa da map olunur.
* Builder.Services.AddAutoMapper(typeof(Program).Assembly) yəni bu proqramdakı bütün tipləri tanısın deyə.
* Imapper interfacei ilə inject edirik. Mapper.Map<City>(dto) göstərməklə gələn dto-nu City classına auto map etmək mümkündür.
* .ForMember(dest=> dest.PhotoUrl, option => { option.MapFrom(src => src.ImageLinks.FirstorDefault(x=>x.IsMain)} );

Lesson 15. JWT Bearer.

* JWT – bir veb tokendir.
* JWT Token Structure – Header Payload və Structure-dən ibarətdir.
* Payload-da həmin user-ə aid olan məlumatları yadda saxlayır. Username role id və s.
* Header-də tokenin tipini hansı şifrələmə alqoritmi ilə yaranıb və s. saxlayır.
* Signature-də isə həmin şifrələmə alqoritminə əsasən tokeni yoxlayır ki həmin alqoritmə əsasən mi yaranıb yoxsa yox.
* JwtBearerDefaults Microsoft.AspNetCore.Authentication.JwtBearer package-indən gəlir.

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey=true,

IssuerSigningKey=new SymmetricSecurityKey(key),

ValidateIssuer=false,

ValidateAudience=false,

};

});

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(\_configuration.GetSection("AppSettings:Token")?.Value);

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject=new ClaimsIdentity(new Claim[]

{

new Claim(ClaimTypes.NameIdentifier,user.Id.ToString()),

new Claim(ClaimTypes.Name,user.Username)

}),

Expires=DateTime.Now.AddDays(1),

SigningCredentials=new SigningCredentials(new SymmetricSecurityKey(key),SecurityAlgorithms.HmacSha512Signature)

};

var token=tokenHandler.CreateToken(tokenDescriptor);

var tokenString=tokenHandler.WriteToken(token);

return Ok(tokenString);

Lesson 16 – SignalR

ASP.NET-də SignalR: Tam İcmal və İstifadə Qaydaları

SignalR, ASP.NET üçün real-time (canlı) məlumat ötürməsini təmin edən bir kitabxanadır. Bu texnologiya sayəsində server və müştəri (client) arasında iki istiqamətli (bidirectional) və asinxron (asynchronous) ünsiyyət yaratmaq mümkündür.

1. SignalR nədir və nə üçün istifadə olunur?

SignalR, əsasən aşağıdakı hallarda istifadə olunur:

Canlı Çat (Live Chat) – İstifadəçilər arasında ani mesajlaşma.

Bildiriş Sistemi (Notification System) – Serverdən müştəriyə ani bildirişlər göndərmək.

Oyunlar (Online Games) – Multiplayer oyunlar üçün real-time ünsiyyət qurmaq.

Canlı Yayım (Live Streaming) – Serverdən müştərilərə ani məlumat ötürmək.

Real-time Məlumat Paneli (Live Dashboard) – Məsələn, stok bazarı məlumatları və ya sensor oxunuşları.

2. SignalR necə işləyir?

SignalR server və müştəri arasında birbaşa əlaqə yaratmaq üçün müxtəlif nəqliyyat (transport) texnologiyalarından istifadə edir:

WebSockets – Ən effektiv üsuldur və SignalR bunu ilk seçim kimi istifadə edir.

Server-Sent Events (SSE) – Yalnız serverdən müştəriyə məlumat ötürür.

Long Polling – Əgər WebSockets və SSE işləmirsə, alternativ kimi HTTP sorğularından istifadə edir.

3. ASP.NET-də SignalR Qurulması və İstifadəsi

3.1. Server tərəfdə SignalR konfiqurasiyası

1. SignalR kitabxanasını layihəyə əlavə et:

Terminalda bu əmri icra edin:

dotnet add package Microsoft.AspNetCore.SignalR

2. SignalR hub sinfi yaradın:

Hub, server və müştəri arasında əlaqəni idarə edən əsas sinifdir.

using Microsoft.AspNetCore.SignalR;

using System.Threading.Tasks;

public class ChatHub : Hub

{

public async Task SendMessage(string user, string message)

{

await Clients.All.SendAsync("ReceiveMessage", user, message);

}

}

3. Startup.cs və ya Program.cs faylında SignalR-i konfiqurasiya edin:

ASP.NET Core 6 və yuxarı versiyalar üçün Program.cs faylında aşağıdakı kodu əlavə edin:

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapHub<ChatHub>("/chathub");

app.Run();

Bununla, ChatHub /chathub URL-si altında aktiv olacaq.

3.2. Müştəri (Client) Tərəfdə SignalR İstifadəsi

SignalR müştəri tərəfindən JavaScript, .NET və ya digər dillərlə istifadə oluna bilər. Burada JavaScript nümunəsi göstərilir.

1. SignalR JavaScript kitabxanasını əlavə edin:

Əgər ASP.NET layihənizdə müştəri tərəfdə SignalR istifadə edəcəksinizsə, aşağıdakı CDN-lərdən istifadə edin:

<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/7.0.5/signalr.min.js"></script>

2. Serverə qoşulma və mesaj göndərmə:

<script>

const connection = new signalR.HubConnectionBuilder()

.withUrl("/chathub")

.build();

connection.start().then(() => {

console.log("SignalR bağlantısı yaradıldı.");

}).catch(err => console.error(err));

// Serverdən mesaj qəbul etmək

connection.on("ReceiveMessage", (user, message) => {

console.log(`${user}: ${message}`);

});

// Serverə mesaj göndərmək

function sendMessage(user, message) {

connection.invoke("SendMessage", user, message)

.catch(err => console.error(err));

}

</script>

Bu kod, müştərinin /chathub ünvanına qoşulmasını təmin edir və mesajları qəbul edib göndərməyə imkan yaradır.

4. SignalR API ilə işləmə (C# müştəri)

Əgər müştəri tətbiqi C# (.NET Console App və ya WPF) olacaqsa, aşağıdakı nümunəyə baxın.

1. Lazımi paketi əlavə edin:

dotnet add package Microsoft.AspNetCore.SignalR.Client

2. C# müştəri kodu:

using Microsoft.AspNetCore.SignalR.Client;

var connection = new HubConnectionBuilder()

.WithUrl("https://localhost:5001/chathub")

.Build();

await connection.StartAsync();

Console.WriteLine("SignalR bağlantısı yaradıldı.");

connection.On<string, string>("ReceiveMessage", (user, message) =>

{

Console.WriteLine($"{user}: {message}");

});

// Serverə mesaj göndərmək

await connection.InvokeAsync("SendMessage", "User1", "Salam, dünya!");

Console.ReadLine();

5. SignalR Authentication və Avtorizasiya

Əgər yalnız müəyyən istifadəçilərin mesaj göndərməsini istəyirsinizsə, authentication istifadə edə bilərsiniz.

1. Server tərəfində Authentication əlavə edin:

**[Authorize]**

**public class ChatHub : Hub**

**{**

**public async Task SendMessage(string message)**

**{**

**var user = Context.User.Identity.Name;**

**await Clients.All.SendAsync("ReceiveMessage", user, message);**

**}**

**}**

2. Müştəri tərəfində token əlavə edin:

var connection = new HubConnectionBuilder()

.WithUrl("/chathub", options =>

{

options.AccessTokenProvider = () => Task.FromResult("BearerToken");

})

.Build();

Bu halda, yalnız autentifikasiya olunmuş istifadəçilər serverə mesaj göndərə bilər.

6. SignalR-in Yoxlanılması və Debug Edilməsi

SignalR-i test edərkən aşağıdakı metodlardan istifadə edə bilərsiniz:

Chrome DevTools (Network → WS tabı) – WebSocket trafiki yoxlamaq üçün.

Postman və ya Swagger – SignalR API çağırışlarını test etmək üçün.

Logging və Debugging – ASP.NET-də ILogger istifadə edərək SignalR loqlarını aktivləşdirin.

7. Son Nəticə və Nəticə

SignalR real-time ünsiyyət üçün güclü bir texnologiyadır və ASP.NET Core ilə problemsiz inteqrasiya olunur.

Əgər canlı çat, bildiriş sistemi, real-time dashboard və ya oyunlar qurmaq istəyirsinizsə, SignalR ideal seçimdir.

Lesson 17 –

* SignalR nədir?
* Hansı tətbiqlərdə istifadə oluna bilər?
* SignalR quraşdırmaq üçün nə etmək lazımdır? Hub-dan miras alan klas yaradılmalıdır. Methodlar yazılmalıdır. Serviceslərdə əlavə etmək lazımdır.
* Override olunan 2 method var. OnConnected OnDisconnected methodları
* Clients.All.SendAsync edib method göndərəndə həmin method arada qəbul olunmalıdır? Client tərəfdə həmin methoda subscribe olunmalıdır. Connection.On ilə
* Client tərəfdən serverdəki hər hansı methodu işə salmaq üçün isə client tərəfdə connection.İnvoke ilə çağırmalıyıq.
* CORS issue nədir? Brauzerdən gələn sorğuları filterləmək üçün istifadə olunur.
* Groups.AddToGroupAsnyc(Context.ConnectionId, “roomName”) ile connectiondaki useri room-a elave edirik
* Clients.OthersInGroup(roomname).SendAsync() etmekle groupdaki ozunden basqa userlere mesaj gonderir

Lesson 22

Mikroservislər (Microservices) — proqram təminatını kiçik, müstəqil və bir-biri ilə API-lər vasitəsilə əlaqə quran modullara bölən bir arxitektura modelidir. Bu yanaşmada hər mikroservis müəyyən bir funksiyanı yerinə yetirir və digər mikroservislərlə minimal asılılıqla işləyir.

**Mikroservislərin Əsas Xüsusiyyətləri:**

1. **Müstəqillik** – Hər mikroservis ayrıca işləyir və inkişaf etdirilə bilər.
2. **Ölçüləbilənlik (Scalability)** – Yalnız yüksək yüklənən hissələr ayrıca genişləndirilə bilər.
3. **Çeviklik (Agility)** – Komandalar müxtəlif texnologiyalardan istifadə edərək mikroservisləri inkişaf etdirə bilər.
4. **Texnologiya müstəqilliyi** – Hər mikroservis fərqli proqramlaşdırma dili və verilənlər bazasından istifadə edə bilər.
5. **Xətalara qarşı dayanıqlılıq** – Bir mikroservisdə problem yaransa, digər mikroservislərin işləməsinə təsir etməz.

**Mikroservis Arxitekturasının Üstünlükləri:**

* Daha asan texniki xidmət və yenilənmə imkanı
* Böyük komandalar üçün daha rahat inkişaf mühiti
* Sərbəst texnologiya seçimi və çeviklik

**Mikroservislərin Dezavantajları:**

* Kompleks idarəetmə və yerləşdirmə (deployment)
* Mikroservislər arasında əlaqə və koordinasiya çətin ola bilər
* Monitorinq və debugging daha çətin ola bilər

Mikroservislər əsasən **bulud əsaslı sistemlərdə**, **böyük və mürəkkəb layihələrdə**, **dağıtılmış sistemlərdə** istifadə olunur. Netflix, Amazon və Uber kimi şirkətlər bu arxitekturanı aktiv şəkildə tətbiq edirlər.

Lesson 23

**ASP.NET Web API-də Request Constraint-lər nədir və necə istifadə olunur?**

**Request Constraint-lər**, ASP.NET Web API-də xüsusi şərtlər (məsələn, route parametrlərinin dəyərləri, HTTP metodları və s.) əsasında fərqli controller və ya action metodlarının seçilməsini təmin edən qaydalardır. Bunun sayəsində eyni route pattern-ə malik olan müxtəlif metodlar arasından düzgün olanı seçmək mümkündür.

**1. Request Constraint növləri**

ASP.NET Web API-də bir neçə növ constraint mövcuddur:

1. **Built-in (daxili) constraint-lər**
2. **Custom (xüsusi) constraint-lər**

### **2. Built-in Constraint-lər**

ASP.NET Core, bəzi hazır query və route constraint-ləri dəstəkləyir. Aşağıda bəzi daxili constraint-lər verilmişdir:

| **Constraint** | **Təsvir** | **Nümunə** |
| --- | --- | --- |
| int | Parametrin tam ədəd olmasını tələb edir | {id:int} |
| bool | Parametrin boolean (true / false) olmasını tələb edir | {status:bool} |
| datetime | Parametrin DateTime formatında olmasını tələb edir | {date:datetime} |
| guid | Parametrin GUID formatında olmasını tələb edir | {id:guid} |
| length(x,y) | Parametrin uzunluğunu təyin edir | {name:length(3,10)} |
| minlength(x) | Parametrin minimum uzunluğunu təyin edir | {name:minlength(5)} |
| maxlength(x) | Parametrin maksimum uzunluğunu təyin edir | {name:maxlength(15)} |
| range(x,y) | Parametrin müəyyən aralıqda olmasını tələb edir | {age:range(18,65)} |
| alpha | Yalnız hərflərdən ibarət olmasını tələb edir | {username:alpha} |

**Nümunə:**

[Route("api/products/{id:int}")]

public IActionResult GetProductById(int id)

{

return Ok(new { Id = id, Name = "Product" + id });

}

Bu API yalnız tam ədəd (int) qəbul edən id ilə işləyəcəkdir.

**3. Custom Constraint-lərin yaradılması**

Əgər daxili constraint-lər sizin tələblərinizə uyğun deyilsə, öz xüsusi constraint-lərinizi yarada bilərsiniz.

**Addım 1: Constraint interfeysini implement edin**

Custom constraint yaratmaq üçün IRouteConstraint interfeysini implement edirik:

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Routing;

public class EvenNumberConstraint : IRouteConstraint

{

public bool Match(HttpContext? httpContext, IRouter? route, string routeKey,

RouteValueDictionary values, RouteDirection routeDirection)

{

if (values.ContainsKey(routeKey) && int.TryParse(values[routeKey]?.ToString(), out int number))

{

return number % 2 == 0; // Cüt ədədləri qəbul et

}

return false;

}

}

**Addım 2: Constraint-i Route-a əlavə edin**

Startup.cs-də (və ya .NET 6+ üçün Program.cs-də) constraint-i qeydiyyatdan keçirin:

using Microsoft.AspNetCore.Routing;

var builder = WebApplication.CreateBuilder(args);

// Custom Constraint əlavə edilir

builder.Services.Configure<RouteOptions>(options =>

{

options.ConstraintMap.Add("even", typeof(EvenNumberConstraint));

});

var app = builder.Build();

app.MapControllers();

app.Run();

Addım 3: Route-da Constraint-dən istifadə edin

[Route("api/numbers/{id:even}")]

public IActionResult GetEvenNumber(int id)

{

return Ok($"Cüt ədəd qəbul edildi: {id}");

}

Bu API yalnız **cüt ədədləri** (2, 4, 6, ...) qəbul edəcəkdir.

**4. Query String Constraint-lər**

Bəzən bir API metodunu yalnız müəyyən query string dəyərlərinə görə seçmək istəyə bilərsiniz. Bunun üçün [HttpGet] atributu ilə Route-da ? işarəsindən sonra şərtlər əlavə edə bilərsiniz.

[HttpGet("api/orders", Name = "GetOrdersByStatus")]

public IActionResult GetOrdersByStatus([FromQuery] string status)

{

if (status == "pending")

{

return Ok("Sifarişlər gözləmədədir.");

}

else if (status == "completed")

{

return Ok("Tamamlanmış sifarişlər.");

}

return BadRequest("Yanlış status parametri.");

}

/api/orders?status=pending və ya /api/orders?status=completed şəklində çağırıla bilər.

**Nəticə**

Request Constraint-lər, ASP.NET Web API-də **route parametrlərinin məhdudlaşdırılması** və **dəqiq uyğunluğun təmin olunması** üçün istifadə olunur.

* **Daxili constraint-lər** (int, bool, guid və s.) API-lərin daha dəqiq işləməsinə kömək edir.
* **Custom constraint-lər** xüsusi ehtiyacları ödəmək üçün istifadə olunur (məsələn, yalnız cüt ədədləri qəbul etmək).
* **Query string constraint-ləri** müəyyən URL parametrlərinə uyğunluğu məhdudlaşdırır.

**Action Filter-lər**, ASP.NET Web API-də **controller və ya action metodlarından əvvəl və sonra xüsusi kod icra etməyə imkan verən** filtrlərdir. Bu filtrlər əsasən **authentication (identifikasiya), logging (jurnal yazıları), error handling (səhv emalı), caching (keşləmə)** və digər məqsədlər üçün istifadə olunur.

**1. Action Filter-lərin Növü**

ASP.NET Core-da aşağıdakı növ filterlər mövcuddur:

1. **Authorization Filters** - API-ə giriş icazəsini yoxlayır.
2. **Resource Filters** - Request işləndikdən əvvəl və sonra icra olunur.
3. **Action Filters** - Controller metodlarından **əvvəl və sonra** icra olunur.
4. **Exception Filters** - Səhvləri emal etmək üçün istifadə olunur.
5. **Result Filters** - ActionResult icra edildikdən sonra nəticəni dəyişmək üçün istifadə olunur.

Bu mövzuda **Action Filters**-ə fokuslanacağıq.

**2. Action Filter necə yaradılır?**

Action Filter yaratmaq üçün ActionFilterAttribute sinfindən irs almaq lazımdır.

**Nümunə: Log yazan Action Filter**

Aşağıdakı nümunədə hər bir **API çağırışından əvvəl və sonra log** yazan bir filter yaradılır:

using Microsoft.AspNetCore.Mvc.Filters;

using Microsoft.Extensions.Logging;

using System;

public class LogActionFilter : ActionFilterAttribute

{

private readonly ILogger<LogActionFilter> \_logger;

public LogActionFilter(ILogger<LogActionFilter> logger)

{

\_logger = logger;

}

// Action başlamazdan əvvəl icra olunur

public override void OnActionExecuting(ActionExecutingContext context)

{

\_logger.LogInformation($"Action başladı: {context.ActionDescriptor.DisplayName} - {DateTime.Now}");

}

// Action bitəndən sonra icra olunur

public override void OnActionExecuted(ActionExecutedContext context)

{

\_logger.LogInformation($"Action bitdi: {context.ActionDescriptor.DisplayName} - {DateTime.Now}");

}

}

Bu filter action başlamazdan əvvəl və sonra log yazacaq.

**3. Action Filter-dən necə istifadə olunur?**

**(1) Filter-i Action metoduna tətbiq etmək**

Filtrimizi **müəyyən bir API metoduna tətbiq edə bilərik**:

[ServiceFilter(typeof(LogActionFilter))]

[HttpGet("api/products")]

public IActionResult GetProducts()

{

return Ok(new { Message = "Məhsullar siyahısı" });

}

**(2) Filter-i Controller səviyyəsində tətbiq etmək**

Əgər filter-i bütün **controller metodlarına** tətbiq etmək istəyiriksə:

[ApiController]

[Route("api/[controller]")]

[ServiceFilter(typeof(LogActionFilter))]

public class ProductsController : ControllerBase

{

[HttpGet]

public IActionResult Get()

{

return Ok("Məhsullar siyahısı");

}

[HttpPost]

public IActionResult Create()

{

return Ok("Məhsul yaradıldı");

}

}

**Bu zaman bütün metodlar üçün filter işləyəcəkdir.**

**(3) Global səviyyədə filter tətbiq etmək**

Əgər filter-in bütün API metodları üçün keçərli olmasını istəyirsinizsə, Program.cs-də qeydiyyatdan keçirmək lazımdır:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>

{

options.Filters.Add(typeof(LogActionFilter)); // Bütün API metodlarına tətbiq edilir

});

var app = builder.Build();

app.MapControllers();

app.Run();

Bu metod, filter-i bütün controller-lərə və action-lara tətbiq edir.

**4. Action Filter-lərin Praktik İstifadəsi**

**(1) Request və Response manipulyasiyası**

Məsələn, hər bir API cavabına xüsusi header əlavə etmək üçün filter-dən istifadə edə bilərik:

public class AddCustomHeaderFilter : ActionFilterAttribute

{

public override void OnResultExecuting(ResultExecutingContext context)

{

context.HttpContext.Response.Headers.Add("X-Custom-Header", "Bu header filter tərəfindən əlavə olundu");

}

}

İstifadəsi:

[ServiceFilter(typeof(AddCustomHeaderFilter))]

[HttpGet("api/data")]

public IActionResult GetData()

{

return Ok("Data ilə xüsusi header göndərilir");

}

**Bu filter hər cavaba "X-Custom-Header" header-ini əlavə edəcəkdir.**

**(2) API cavablarını loglamaq**

Aşağıdakı nümunədə hər API cavabını loglayan filter yaradılıb:

public class ResponseLoggingFilter : ActionFilterAttribute

{

private readonly ILogger<ResponseLoggingFilter> \_logger;

public ResponseLoggingFilter(ILogger<ResponseLoggingFilter> logger)

{

\_logger = logger;

}

public override void OnActionExecuted(ActionExecutedContext context)

{

\_logger.LogInformation($"Response göndərildi: {context.HttpContext.Response.StatusCode}");

}

}

Bu filter API-nin cavab status kodlarını loglayacaq.

**5. Nəticə**

Action Filter-lər **ASP.NET Web API-də kodu təkrarlamadan müəyyən əməliyyatları avtomatik icra etməyə** imkan yaradır.

✅ **Əsas istifadə halları:**

* **Loglama və monitorinq**
* **Səhv emalı və exception handling**
* **Cavablara header-lər əlavə etmək**
* **Validasiya və icazə yoxlamaları**