**Create ASP.NET MVC Project**

Visual Studioda MVC project yaratmaq üçün ASP.NET Core Web App və ya ASP.NET Core Empty project yaratmaq lazımdır. Web App project daxilində MVC folder structure hazır şəkildə gəlir. Lakin Empty project yaratsaq özümüz bu MVC structure’ni qurmalıyıq.

**MVC Structure**

Model’lərimiz Models folder’i daxilində, View’larımız Views folder’i daxilində və sonda Controller’lərimiz isə Controllers folder’i daxilində olmalıdır. Structure’in düzgün işləməsi üçün (Assembly’in düzgün oxuya bilməsi üçün) View’larımızın sonu View, Controller’lərimiz isə Controller ilə bitməlidir və Controller class’ı extend etməlidir.

**Services - AddControllerWithViews()**

Biz yaratdığımız project’də MVC structure’dan istifadə edəcəyimizi bildirmək üçün Program.cs daxilində builder.Services.AddControllerWithViews(); yazmaq lazımdır. Bu şəkildə MVC project olduğu Assembly tərəfindən başa düşülür.

**Middleware - MapControllerRoute()**

Middleware hissəmiz isə request’ləri düzgün qəbul edilib, idarə edilə bilməsi üçün istifadə edilir. Başlanğıc olaraq biz middleware hissədə app.MapControllerRoute method’dan istifadə etməliyik. Arqument olaraq isə name arqumentinə qarşılıq olaraq “default” dəyəri, pattern aqrumentinə qarşılıq olaraq isə “{controller=Home}/{action=Index}/{id?}” göndərilməlidir. Burada biz göstəririk ki, path’də domain name’dən sonra /controllerName/actionName gələcək, gəlməsə default olaraq controller Home, onun action’u olaraq isə Index işə düşəcək. Nullable olaraq isə Id də gələ bilər.

**Razor View - View Start**

View’lar, Razor View Empty və Razor View file’lar olaraq 2 cür olur. Razor View Empty-nin fərqi odur ki, Razor View yaradan zaman biz hər hansısa layout’un istifadə olunub-olunmayacağını qeyd edə bilərik. Əgər istifadə olunacaqsa Layout’un ünvanını göstərməliyik. Bu şəkildə Razor View yaranan zaman ən üst hissədə Layout = \_Layout avtomatik olaraq qeyd ediləcək.

Layout’dan istifadə edəcəyimizi qeyd edib, hər hansısa ünvan göstərməyib boş buraxsaq o zaman Razor View Start file’ın olduğu anlaşılır. Çünki View folder daxilində yaradılan Razor View Start file’ı daxilində, Layout’dan istifadə olunacağı qeyd edilən Razor View’un hansı Layout’u istifadə edəcəyini göstərmək üçün istifadə edilir və yuxarıdaki kimi View yaranan zaman avtomatik olaraq Layout = \_Layout yazılmayacaq.

**Shared folder - Layout**

Bizim yaratdığımız View’lar daxilində bir sıra hissələr (məs. header,footer) təkrarlanan kodlar olacaq. Bu təkrarın qarşısını almaq üçün Views folder daxilində Shared folder yaradılmalıdır. Shared folder daxilində Razor Layout file’ı saxlayır.

**View Model**

Controller daxilindəki Action’larda return View() edən zaman biz lazım olan model’i view’a(front’a) göndərə bilərik. Lakin bəzən biz eyni anda bir neçə model’i göndərməli ola bilərik. Bununla yanaşı biz security cəhətdən model’imizin bütün property’lərini view’a(front’a) göndərməməliyik. Bu problemləri aradan qaldırmaq üçün View Model anlayışı yaranır. Yaratdığımız View Model class’ların sonu VM və ya ViewModel ilə bitməsi daha anlaşıqlı olması üçün idealdır. Bu VM class’lar daxilində həmin View’a göndərməli olduğumuz bütün object’ləri property olaraq qeyd etməyimiz və View’a bu VM’i göndərməyimiz kifayət edir.

**MVC Data Passing - ViewBag,ViewData and TempData**

Data Passing, ASP.NET MVC və ASP.NET Core MVC daxilində controller’dən view’a və ya eyni controller daxilindəki fərqli action’lar arasında(TempData) data transferi üçün nəzərdə tutulmuşdur. ViewBag ilə ViewData bir-birlərinin alternatividir.

ViewBag C#’daki dynamic type’ı istifadə edir, yəni compile time yox run time dəyər mənimsədilir. Bu səbəbdən bu səbəbdən düzgün şəkildə idarə olunmasa runtime errorlara gəlib çıxa bilər. ViewBag.PropertyName yazaraq biz adını PropertyName olaraq qeyd etdiyimiz datanı çağırırıq, ViewBag.PropertyName = value; yazaraq PropertyName’ə value mənimsədirik. Burada value compiletime yox runtime mənimsədilir.

ViewData[“PropertyName” ] yazaraq biz adını PropertyName olaraq qeyd etdiyimiz datanı çağırırıq, ViewData[“PropertyName” ] = value; yazaraq PropertyName’ə value mənimsədirik.

Əgər biz return View() əvəzinə return RedirectToAction(“ViewName”) yazsaq ViewBag ilə ViewData qeyd edilən dataları saxlamayacaq. RedirectToAction edən zaman dataların saxlanılmasını istəyiriksə yeni bir anlayışla tanış olmalıyıq. Bu TempData’dır.

TempData[“PropertyName” ] yazaraq biz adını PropertyName olaraq qeyd etdiyimiz datanı çağırırıq, TempData[“PropertyName” ] = value; yazaraq PropertyName’ə value mənimsədirik. ViewData ilə eyni syntax’la data mənimsədilir və ya çağırılır. Fərqi isə yuxarıda izzah etdiyimiz kimidir.

**Return View(), RedirectToAction(), ViewName() or View(“ViewName”)**

Hər hansısa Controller’in Action’ı(method’u) işə salınan zaman return olaraq IActionResult interface’dən miras alan bir Method return olunmalıdır. Bunun isə 3 yolu mövcuddur.

Return View() edən zaman assembly tərəfindən başa düşülür ki, View>ControllerName>ActionName.cshtml mövcuddur və geriyə ActionName.cshtml qaytarılır.

Return ViewName(); və ya return View(“ViewName”); , View’nu renderləmək, render olunan view’un HTML’ni hal-hazırdaki request’ə response olaraq göndərmək üçün istifadə edilir. Method işə düşən zaman return’a qədər olan hissə nəzərə alınmır. ViewBag, ViewData və ya TempData buna daxildir. Daha sonra redirect olunan Action’ın içərisi oxunur və response olaraq həmin Action’a uyğun olaraq View’su göndərilir.

Əsasən User’i başqa bir səhifəyə(URL) yönləndirmək üçün istifadə edilən

return RedirectToAction(“Action Name”); , Contoller daxilindəki bir Action’a və ya başqa bir Controller’in Action’na yönləndirmə, yəni HTTP redirect etmək üçün istifadə edilir. Bu method browser’ə HTTP 302 response göndərəcək, bu halda response browser’ə redirect olunan action üçün yeni bir GET request’i baş verməsini tələb edəcək. Bununla da əvvəlki URL redirect olunan URL’lə əvəz olunacaq.

**Database**

Biz səhifədə göstərməli olduğumuz dataları Database daxilində saxlayırıq. Database’lər isə bizim məqsədimizə və ya istəyimizə uyğun olaraq dəyişir.

**DbContext**

İlk olaraq DbContext class’dan miras alan AppDbContext class’ı yaratmalıyıq. DbContext’dən miras ala bilmək üçün NuGet Packages’dən Microsoft.EntityFrameworkCore package yüklənməlidir. AppDbContext class daxilində Constructor yaradılmalı və parametr olaraq DbContextOptions<AppDbContext> type’dan olan options object’i qəbul etməlidir daha sonra base class’a(DbContext class’a) options object’i göndərməlidir.

**DbSet<T>**

Beləliklə biz Database daxilində yaranmasını istədiyimiz Table’lar üçün AppDbContext class daxilində generic olan public DbSet<EntityName> EntityNames {get;set;} property’ləri yaratmalıyıq. Yaratdığımız hər bir DbSet property Database daxilində bir table’a qarşılıq gəlir bu səbəbdən generic type olaraq model class’larımızı yazırıq.

**AddDbContext**

Biz Program.cs daxilində Services hissədə Dependency Injection baş verə bilsin və Database ilə əlaqə yarana bilsin deyə bildirməliyik ki DbContext’dən istifadə etmişik. builder.Services.AddDbContext<AppDbContext>(opt=>{ opt.UseSqlServer(connectionString) });

**UseSqlServer**

UseSqlServer’dən istifadə edə bilməyimiz üçün

NuGet Packages’dən MsSql üçün istifadə olunan Microsoft.EntityFrameworkCore.SqlServer package yüklənməlidir.

**Connection string**

Connection string məxfi data olduğu üçün onu kənardan çağırmalıyıq. appsettings.json file daxilində json şəklində saxlayacayıq. Connection string aşağıdaki şəkildə olmalıdır.

“ConnectionStrings” :{“Default” : “Server:ServerName;Database=DatabaseName;Trustesd\_Connection=True;”}

.cs file daxilində bir neçə yolla çağrıla bilər. Bunlardan biri Configuration class’dan istifadə etməkdir. builder.Configuration[“Key:Value”] məntiqi ilə çağırıla bilər. builder.GetConnectionString(“Value”); yazaraq da eyni işi görə bilərik.

**Migrations**

Biz code hissədə etdiyimiz dəyişikliklərin Database’də öz əksini tapa bilməsi üçün terminal olaraq həm Package Manager Console’dan həm də Developer PowerShell’dən istifadə edə bilərik. Lakin NuGet Packages vasitəsi ilə Microsoft.EntityFrameworkCore.Tools package yüklənməlidir. Package Manager Console’da migration’ları yaradan zaman Default Project olaraq AppDbContext’in yerləşdiyi layer’ı seçirik. Migration’ların yarana bilməsi üçün əlavə olaraq Microsoft.EntityFrameworkCore.Design package yüklənməlidir.

add-migration “MigrationName” yazaraq edilən dəyişiklikləri yeni yaratdığımız migration daxilinə əlavə edirik.

update-database vasitəsi ilə biz yaratdığımız yeni migration’u database’ə əlavə edirik.

update-database MigrationName vasitəsi ilə biz database’i daha əvvəldən yaratdığımız migration halına gətirə bilərik.

remove-migration vasitəsi ilə update olunmamış migration’u silirik. Update olunan migration’u silmək istədiyimiz halda isə ondan əvvəlki migration’a update edərək silmək istədiyimiz migration’u remove-migration yazaraq silirik. Manual olaraq gedib migration folder daxilindən migration’u silmək məsləhət görülmür.

**Validate Anti Forgery Token**

Form əməliyyatları zamanı hər hansısa hack hücumları(XSRF/CSRF attacks) həyata keçə bilər. Role yoxlanışı ilə yanaşı həm də biz third part bir app (məs. Postman) vasitəsilə bizim application’a request göndərməyin qarışısını almaq üçün Controller’in method’u üçün [ValidateAntiForgeryToken] Annotation’dan istifadə etməliyik. Bu şəkildə artıq bizim app’dən başqa heç bir app’dən bu ünvana request atıla bilməyəcək.

**AutoMapper**

Biz security cəhətdən project daxilində Front’a database’dən gələn Entity object’ləri bir başa olaraq göndərməyimiz təhlükəlidir. Çünki biz bu object’ləri bir başa ötürsək əlçatan olmamalı bir sıra dataları da (property’ləri) göndərmiş olarıq. Bunun qarşısını almaq üçün ViewModel və ya DTO class’ları yaradıb müəyyən olunmuş property’ləri saxlayırıq.

Hər dəfə datanın ötürülməsi və ya qəbul olunması zamanı yeni ViewModel & DTO və ya Entity object yaradılması və dəyərlərin mənimsədilməsi həm kod təkrarına gətirib çıxarır həm də vaxt itkisinə. Bu problemin qarşısını isə reflection ilə rahatlıqla ala bilərik. Lakin bunu bizim üçün əvvəlcədən edib, AutoMapper package halına salıblar.

AutoMapper və AutoMapper.Extensions.Microsoft.DependencyInjection package yüklənlənməlidir. Daha sonra hər Entity adına uyğun olaraq AutoMapper class yaradılmalı, Package vasitəsilə gələn Profile class’dan miras almalıdır. Boş constructor override edilməli, CreateMap<T,R>(); generic method işə salınmalıdır. Bu halda göndərilən T type(class) və R type(class) daxilindəki property’lər və onların value’ları Assembly tərəfindən oxunur, T type daxilindəki property’lərdən adları R type daxilindəki property’lər ilə üst-üstə düşənlər yeni R type object’ə mənimsədilir və geriyə qaytarılır. Burada adları eyni lakin datatype’ları fərqli olan property’lər də ola bilər. Bu halda runtime exception baş verəcək.

ReverseMap(); method’dan da istifadə edərək həm Entity göndərib ViewModel & DTO object’i həm də ViewModel & DTO göndərib Entity objecti qəbul edə biləcəyimizi bildiririk.

Sonda services olaraq adlandırdığımız hissədə gəlib builder.Services.AddAutoMapper(typeof(mapperAssembly)); vasitəsilə Mapper class’ların yerləşdiyi Assembly’ni tanıdaraq prosesi yekunlaşdırırıq.

Dependency injection ilə Mapper’dan istifadə edəcəyimiz class daxilində IMapper object’i çağırırıq, Map<R>(T); method’u tətbiq edib R type object’i alırıq.

**File Upload**

Yazılacaq proyektlər daxilində istəyə uyğun olaraq hər hansısa bir fayl əlavə etmək funksiyası ola bilər. Biz fayllara qarşılıq olaraq C#’da IFormFile’dan istifadə edəcəyik. Və artıq input type file olaraq yüklənən bütün file’ları IFormFile ilə qəbul edə bilərik.

MVC proyektdə Tag Helper’in köməkliyi ilə input daxilində asp-for’a yazılan property’in type’ı IFormFile olarsa, input’un type’ı avtomatik olaraq file başa düşülür. Lakin biz form’u submit etdiyimiz zaman fayl seçdiyimiz halda bu property null olaraq qayıdacaq. Bu situasiyanın yaşanmaması üçün kiçik bir dəyişiklik etməliyik. Form tag’i daxilində enctype = “multipart/from-data” yazaraq problemi aradan qaldırırıq.