# **NOTLAR**

* .Net Core teknolojisinin sunuluş amaçlarından birisi ve hatta en önemlisi platformdan bağımsız olarak çalışabilmesidir.
* .NET Core ile birlikte gelen Kestrel diğer web sunuculardan bağımsız kendi web sunucu özelliklerini barındıran bir yapıdır. Bu bağımsızlığı sayesinde cross-platform uygulamalar geliştirme altyapısı sunar.
* Kestrel .Net Core’ un daha hızlı çalışabilmesini sağladı.
* Main metotu içerisindeki CreateHostBuilder kestrel web serverını ayağa kaldırabilen bir metottur. Varsayılan bir yapı olarak gelir.
* Web Server ile ilgili ayarlamaları yapabileceğimiz bir Startup sınıfı da konfigürasyon dosyası olarak kullanılıyor.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

* .Net Core’ un .Net’ ten daha hızlı çalışmasının sebebi Kestrel Web Server ve Configuration’ dur. Kestrel Web Server oldukça hafif bir sunucudur.
* .Net Core uygulamaları açık kaynak kodludur.
* UseRouting: Yukarıdaki url’ yi çalıştırıyor. Bunu kaldırırsan url’ nin bir anlamı kalmıyor. // ysk.com.tr/
* UseEndPoints: Kaldırılınca uygulama çalışır ama herhangi bir tepki vermez. Url’ ye karşılık gelen bir şey bulamıyor. Biz bu sayfayı bulamadık diyor.
* Response(tepki, cevap) ve Request(istek) olaylarını kontrol etme durumuna middleware deniyor. Özünde ara katman demek. Response ve request arasında işlem görüyor.
* Delegeler sırasıyla metotları tetikleyen yapılardır. Bir metotu ya da metotlar kümesini çalıştırır.
* Invoke kullanıldığında ise delegenin işaret ettiği ne kadar metot varsa hepsi sırasıyla çalışacak.
* Asenkron mimari birbirini bloklamayan kod kümelerini yazmamıza olanak sağlıyor. Mesela ben bir toplama işlemi ve çarpma işlemi yaptım toplama işlemini anında yapmak istiyorum ama çarpma işlemim 10 saniye sonra olsun istiyorum. Eğer senkron mimari olursa çarpma işlemi bitene kadar hiçbir şey yapamam ve işlemlerim bloklanır, süre bitene kadar beklemem gerekir. Asenkron mimari oldduğunda çarpma işlemi sonucu çıkana kadar istediğimiz işlemleri yapabiliriz, bizi engellemez veya bloklamaz. İşlem bittiğinde de ekranda mesaj kutusunda sonucu görürüz.
* Asenkronik metot yapacaksak eğer async demeliyiz ve parantez içindeki kodun önüne await koymalıyız.

Örnek;

private async void btnCarp\_Click(object sender, EventArgs e)

{

await CarpAsync(int.Parse(textBox1.Text), int.Parse(textBox2.Text));

}

* async ilgili işi asenkronik hale getiriyor.
* await ise ilk işlem sonuçlanana kadar ikincisine geçme diyor. Yani önce işlem yapılacak, çarpma işlemi olacak o ekrana geldikten sonra ise işlem gerçekten tamamlandığında işlem tamamlandı diyecek. Eğer await koymazsak çarpma işlemi arkada devam eder ama kod çalışır çalışmaz çarpma işlemi henüz sonucunu vermeden ekrana işlem tamamlandı mesajı gelir. Kısacası await koyduğumuzda işlemlerin sıralı bir şekilde gitmesini sağlıyoruz.

Örnek;

private async void btnCarp\_Click(object sender, EventArgs e)

{

await CarpAsync(int.Parse(textBox1.Text), int.Parse(textBox2.Text));

MessageBox.Show(“İşlem tamamlandı”);

}

* action delege: Herhangi bir tipten değer alıp geriye herhangi bir şey döndürmeyen delegelerdir.
* Delege yazacağımız zaman delagate keywordünü kullanmalıyız.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Geriye döndüreceğimiz değeri task içerisinde belirtiyoruz.

Örnek;

private Task< int > CarpAsync(int sayi1, int sayi2)

* Tasktan sonra int vs. diyip sonuç döneceksem parantezin içinde return demeliyim.

Örnek;

private Task< int > CarpAsync(int sayi1, int sayi2)

{

return Task.Run(() =>

{

Thread.Sleep(10000);

return sayi1\* sayi2;

});

}

* Eğer bir şeyde sadece task varsa o onun void olduğunu gösterir.
* MVC Asp.Net’ ten önce Java, Php gibi dillerde de kullanılmıştır. Model-View-Controller.
* View: İlgili kullanıcının gördüğü kısmı temsil eder.

Controller: Viewleri kontrol eden yapıdır.

Model: İlgili nesnelerdir.

* Mesela bir veriyi çektiğim zaman bu veriyi bir modele bind ederim ve controller aracılığıyla ilgili viewe gönderirim.
* İlgili controllerımın içerisindeki actiondan bir datayı ilgili viewimde göstermek istiyorum. Bunun için 4 yöntem var.

ViewBag: ViewBag.Name = “Yavuz”;

ViewData: ViewData[“Name”] = “Selim”;

TempData: TempData[“Name”] = “Kahraman”; TempData’ nın actionlar arasında veri taşınabilir.

Model: Bunun için bir modele ihtiyacımız var. Models’ da bir class oluşturup içine propertylerimizi giriyoruz. Controllerda tanımlayıp dolduruyoruz ve view sayfasının başında @model diyerek tanımlama yapıyoruz.

* Bir interface örneklenemez fakat nesne örneği taşıyabilir. (Kendisinden kalıtılan nesne örneğini.)

Örnek;

ITest test = new ITest(); Yapamayız hatalı olur.

ITest test = new Test(); Yapabiliriz hata olmaz.

IActionResult da aynı şekilde örneklemez ama içerisinde örnekler barındırabilir.

* RouteData ile bütün route lara erişebiliriz.
* [Route(“Yavuz”)] kodunu IActionResult’ ın üzerine yazarsak eğer bu sadece Yavuz yazıldığında bu sayfayı çalıştır anlamına gelir. Ve Startup sınıfındaki kodları ezerek onları geçersiz hale getirir. Sadece Yavuz kelimesi kullanıldığında sayfa açılır.
* Attribute bazlı yazmış olduğumuz routing startup tarafındaki kodları ezer.
* app.UseStaticFiles(); dendiğinde wwwroot klasörünün erişimi dışarıya açılır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

* node modules path’i istendiğinde fileprovider’ daki dosyayı aç. Dosyayı dışarıya erişim vermek için bu kod yazılır. Biz burda bootstrap, jquery, css gibi dosyalarımızı dışarıya erişim vermek için bu kodu yazdık. (npm ile paket yükleme yolu).
* PredicateDelegate: String bir değer alıp geriye bool değer döndürüyor. String bir listede istenen değeri bulmaya kolaylık sağlar.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Predicate delegeyi yorum satırlarında olduğu gibi de oluşturabiliriz veya liste şeklinde de oluşturabiliriz. Burada verdiğimiz örnekte isimlerin içinden bize Yavuz değerini bularak döndürüyor. Her seferinde isimleri dönüyor Yavuz’ u bulana kadar false değer veriyor ve Yavuz’ u bulduğunda true değer döndürerek bize bulunanIsim’ in Yavuz olduğu sonucunu veriyor.