# **NOTLAR**

* Controllerlar sadece servisleri bilir. Constructorlarında kesinlikle Repository’ i miras almayacaktır.
* FirstOrDefault dediğimizde eğer bu id’ den db de 4 5 tane varsa ilkini döndürür. Ama SingleOrDefault dersek koşulu karşılayan birden fazla satır bulursa geriye hata döner. Yani farkı First geriye bir exception dönmez ama Single geriye bir exception döner.
* Normalde herhangi bir validation kütüphanesi kullanmadan validation işlemleri gerçekleştirebiliyoruz. Çünkü .Net Frameworkle beraber bize hazır validationlar geliyor. Biz herhangi bir entity veya herhangi bir Dto classlarımızda, propertylerinde Required, Range gibi attributeler kullanarak validation işlemleri yapabiliyoruz. Yalnız bu attributeleri kullandığımız zaman bu entitylerimizi veya dto larımızı kirletmiş oluyoruz. Bizim buraları mümkün olduğunca temiz tutmamız lazım. O zaman bizim burada bu validation işlemlerini tamamen ayrı bir yerde yapmamız gerekiyor. Bunun için Fluent Validation kütüphanesi kullanıyoruz.
* Fluent Validation kullandığımız zaman özellikle API tarafta endpointe bir istek yaptığımızda kendi resultını döner. Kendinin dönmüş olduğu bir model var.
* İnt, decimal gibi alanlar her zaman default olarak 0 olarak gelir. Not null dememize gerek yoktur.
* Filterlar bizim controller içerisindeki metodlarımıza gelen requesti müdahale etmek için kullanabiliyoruz. Yani bir metodumuza request gelmeden önce, geldikten sonra; action metot çalışmadan önce, çalıştıktan sonra; sonuç üretilmeden önce action metotta sonuç üretildikten sonra bu yerlerde müdahalelerde bulunabiliyoruz. Filterlar bize bu müdahale imkanını sağlıyor. Kendimiz de filter oluşturabiliriz.
* API ve mvc(yani web) tarafında hazır bir middleware var useExceptionHandler isminde. Bu middleware uygulamada herhangi bir hata fırlatıldığında o hatayı yakalar. Biz bunu custom ederek kendi nesnemizi dönebiliriz. Eğer bu bir mvc uygulaması ise herhangi bir sayfaya yönlendirme yapabiliriz. Yani burada duurm 2’ ye ayrılıyor.

1. API uygulaması ise geriye bir response dönmeliyiz.
2. Mvc uygulaması ise geriye bir sayfa dönmeliyiz. Yani error sayfası diye bir sayfamız olacak ve oraya yönlendireceğiz.

* App ile başlayıp sonrasında use ifadesi kullananların hepsi birer middlewaredir.

Örnek;

app.UseRouting();

* Middleware:

Uygulamaya bir request geldiğinde aşama aşama middlewarelerden geçer. Response’ a dönüşünce de yine aynı şekilde middlewarelere uğrayarak response’ a dönüşür. Yani şöyle diyebiliriz bir middleware’ e request ilk geldiğinde girer bir de response oluşurken girer.

* Mümkün olduğunca az try catch kullanmak bir Best Practices’ dir. Uygulamada çok fazla try catch bulunması uygulamanın aslında performansının kötü olduğu, hataların iyi handler edilemediği anlamına gelir. Mümkün olduğunca az kullanılması gereklidir, çünkü ciddi bir bloklayıcı etkisi vardır. Ne kadar az olursa o kadar performans iyi olur.
* AutoFac kütüphanesi bir Inversion of Control Container olarak geçer. Aynı zamanda DI (Dependency Injection) Container olarak da geçer. Normalde bir Asp .Net Core projesi oluşturduğumuzda bu frameworkün içerisinde Built in olarak bir DI Container gelir. Yani biz napıyoruz Program.cs içerisinde builder.Services’ lar diyerek herhangi bir classın constructor’ ında kullanacağımız interface’ i ve bu interface’ e karşılık gelen classları ekliyoruz.
* Default olarak gelen DI Container’ da Constructor Injection ve Metod Injection yapabiliyoruz iki tane injection yolumuz var. Ya herhangi bir classın constructor’ ında interface’ i geçebiliyoruz. Ya da herhangi bir classın metodunda interface’ i geçerek bu inteface’ e karşılık gelen classı alabiliyoruz.
* AutoFac bu default olarak gelen Built in DI Container’ dan daha yetenekli. Örneğin Built in DI’ ın constructor injection’ ı var, metod injection’ ı var ama property injection’ ı yok. Mesela AutoFac’ te property injection var.
* Diğer bir önemli özellik ise mesela Asp .Net Core’ da dinamik olarak diğer servislere ekleme özelliğimiz yok. Mesela şöyle diyemiyoruz sonu service ile biten tüm interface’ e karşılık gelen sonu service ile biten classların hepsini ekle diyemiyoruz. Ama AutoFac’ te dinamik olarak diğer container’ a nesne ekleme özelliği var. Yani sonu repository ile biten tüm interfacelere karşılık gelen tüm sınıfları ekle diyebiliyoruz.
* Autofac’ te as ile beraber önce class sonra interface yazılır.
* In-memory caching: Yani dataları uygulamamız nerede host ediliyorsa o hostun memorysini kullanacak. Dataları cacheleyeceğiz ve bu sayede clientlardan bize bir istek geldiğinde önce cache’ de var mı kontrol edeceğiz. Eğer cache’de varsa cache’den dönecek eğer yoksa datayı repodan çekip cacheleyip öyle döneceğiz.
* Cacheleyeceğiniz data için çok sık değiştirmeyeceğiniz ama çok sık erişeceğiniz data en uygun cache adayıdır. Çok sık erişebilirsiniz ama çok sık güncellemediğiniz bir data olmalıdır.
* İstek yapabilmek için HttpClient nesneleri kullanılır.