* [HttpGet] → Tüm ürünleri getir
* [HttpGet("{id}")] → Belirli bir ürünü getir
* [HttpPost] → Yeni ürün ekle
* [HttpPut("{id}")] → Ürünü güncelle
* [HttpDelete("{id}")] → Ürünü sil

**Entities**

**C#'ta Entities, genellikle veri modelini temsil eden sınıflardır.  
Yani, veri tabanındaki tabloların veya işlediğin verilerin C#'taki karşılıklarıdır.**

**Mesela bir veri tabanında Kullanicilar diye bir tablo varsa, C#'ta bunun için şöyle bir entity sınıfı oluştururuz:**

**public class Kullanici**

**{**

**public int Id { get; set; }**

**public string Ad { get; set; }**

**public string Soyad { get; set; }**

**public string Email { get; set; }**

**}**

**Repository**

**Repository dediğimiz şey, Entity'lerle (yani verilerle) yapılan işlemleri yöneten sınıflardır.  
Yani bir Kullanici entity'si için, kullanıcılara dair CRUD işlemlerini (Create, Read, Update, Delete) Repository içinde yaparız.**

**Örneğin:**

**public interface IKullaniciRepository**

**{**

**Kullanici GetById(int id);**

**List<Kullanici> GetAll();**

**void Add(Kullanici kullanici);**

**void Update(Kullanici kullanici);**

**void Delete(int id);**

**}**

**Business**

**Business (veya Service katmanı da denir),  
Entity'lerin ve Repository'lerin üstünde duran, iş kurallarını yöneten katmandır.**

**Yani:**

* **"Bir kullanıcı eklenirken e-mail boş olamaz."**
* **"Bir ürün stoğu sıfırsa satışa kapat."**
* **"Sipariş tutarı 500₺'den büyükse %10 indirim uygula."**

**Bu tür kurallar Business katmanında yazılır.**

**Repository sadece ham veri işlemi yapar.  
Business ise mantık ve kurallar koyar.**

**Özetle:**

* **Entity → Sadece veri (özellikler) tutuyor.**
* **Repository → Veri tabanı işlemleri yapıyor.**
* **Business (Service) → İş mantığını, kuralları uyguluyor.**

**Core**

**Core, projede temel yapıları, ortak kuralları ve soyutlamaları (interface'ler, base sınıflar gibi) topladığımız yerdir.  
Hiçbir katmana bağlı olmaz.  
Diğer katmanlar (Entities, Repositories, Business) Core'dan yararlanır.**

**Örnek bir Core içeriği neler olur:**

* **Ortak Interface'ler (IRepository, IService vs.)**
* **Ortak Base Entity sınıfı (mesela hepsinde Id, CreatedDate gibi alanlar ortaksa)**
* **Ortak Result sınıfları (işlemlerin başarılı mı, hatalı mı olduğunu taşımak için)**
* **Ortak Exceptions, Validators vs.**

**Mesela bir IRepository interface'i Core içinde olur:**

**public interface IRepository<T> where T : class**

**{**

**T GetById(int id);**

**List<T> GetAll();**

**void Add(T entity);**

**void Update(T entity);**

**void Delete(T entity);**

**}**

**Sonra sen KullaniciRepository gibi sınıflarda bunu uygularsın.**

TId demek:

"Bu sınıfı kullanırken Id'nin tipi ne olacaksa, onu dışarıdan sen belirle."

Yani Id tipi **int** mi olacak, **Guid** mi olacak, **string** mi olacak — onu sen seçiyorsun!

### Bir örnek:

Diyelim bir Brand entity'si yapıyorsun ve Id tipi int olsun istiyorsun:

public class Brand : BaseEntitiy<int>

{

public string Name { get; set; }

}

Burada Brand sınıfı, BaseEntitiy<int>'ten miras alıyor.  
Yani Id property'si artık **int** türünde olmuş oluyor!

Başka bir örnek:  
Eğer Guid kullanmak istersen:

public class User : BaseEntitiy<Guid>

{

public string UserName { get; set; }

}

Burada Id tipi **Guid** olacak.

### Yani kısaca:

| **TId** | **Anlamı** |
| --- | --- |
| int | Id bir tamsayı olacak |
| Guid | Id rastgele bir benzersiz değer olacak (GUID) |
| string | Id bir yazı olabilir mesela (örnek: özel kodlar) |

**Entity Framework (EF) Nedir?**

**Entity Framework,  
C# projelerinde veritabanı ile kod arasında köprü kuran,  
senin için SQL sorguları yazan, veritabanı işlemlerini kolaylaştıran bir ORM (Object Relational Mapper) aracıdır.**

**Basitçe:**

**Normalde veri tabanına veri eklemek, silmek, güncellemek için SQL sorguları yazarsın.  
EF kullanırsan, SQL yazmadan C# kodlarıyla bu işleri halledersin.**

**Mesela klasik yöntemle SQL yazmak:**

**sql**

**INSERT INTO Brand (Name) VALUES ('Nike');**

**public virtual Brand? Brand { get; set; } bu ne**

**ChatGPT:**

**Çok iyi soru! Şimdi hemen net ve sade bir şekilde anlatıyorum:**

**public virtual Brand? Brand { get; set; }**

**Bu satırın anlamı şöyle:**

| **Kelime** | **Anlamı** |
| --- | --- |
| **Brand** | **Burada başka bir entity'yi (Brand) temsil ediyor.** |
| **virtual** | **EF'nin bu property'yi lazy loading (isteğe bağlı veri çekme) yapabilmesi için.** |
| **? (soru işareti)** | **Bu property null olabilir demek (opsiyonel ilişki var).** |

**Açıklaması:**

**Bu, bir entity içinde başka bir entity'ye olan ilişkiyi gösteriyor.**

**Örneğin bir Car sınıfı içinde Brand var diyelim:**

**public class Car**

**{**

**public int Id { get; set; }**

**public string ModelName { get; set; }**

**public int BrandId { get; set; } // Foreign Key**

**public virtual Brand? Brand { get; set; } // Navigation Property**

**}**

**Burada:**

* **BrandId ➔ Veri tabanındaki bağlantı için kullanılır (yani Car tablosunda BrandId sütunu olur).**
* **Brand ➔ Brand nesnesine erişmeni sağlar. (Mesela arabayı çağırınca marka bilgisini de çağırabilirsin.)**

**Neden virtual yazıyoruz?**

**Entity Framework, virtual keyword'ünü görünce şunu anlar:**

* **"Bu property'yi veritabanından isteğe bağlı olarak çekebilirim."**
* **Yani Lazy Loading yapabilir.**
* **Gerekmedikçe veritabanına ekstra sorgu atmaz, sadece ihtiyaç duyduğunda getirir.**

**(Performans ve esneklik için çok faydalı!)**

**Peki HashSet<Car> nedir?**

**HashSet<T> koleksiyonu, benzersiz öğeler tutan bir koleksiyondur.  
Yani içinde aynı Car nesnesinden birden fazla olmasına izin vermez.**

**HashSet özellikleri:**

* **Benzersiz elemanlar tutar.**
* **Düzen sağlamaz. Elemanların sırası belli değildir.**
* **Elemanlar üzerinde hızlı bir şekilde arama ve ekleme işlemleri yapabiliriz.**

**enum**

**Burada bir enum (yani enumeration / numaralandırılmış tip) tanımladın.**

**enum nedir?**

* **Belli bir grup sabit değeri bir araya toplamak için kullanılır.**
* **Yani bir değişkenin alabileceği değerleri sınırlandırıyorsun.**
* **Kafanda "bu araba ya kiralık olur, ya müsait olur, ya bakımda olur" diyorsun.**
* **Böylece sabit bir liste yaratıyorsun.**

**Car car = new Car();**

**car.State = CarState.Available; // Araba şu anda müsait**

**ya da**

**car.State = CarState.Maintenance; // Araba bakımda**

**Interface nedir?**

**Interface dediğimiz şey,  
bir sınıfın (class'ın) hangi özelliklere ve fonksiyonlara sahip olacağını söyler,  
ama o özelliklerin nasıl çalışacağını söylemez.**

**Yani, kuralları koyar,  
"Bu fonksiyonlar olacak kardeşim!" der,  
ama içini sen doldurursun.**

**🔵 Basit örnek:**

**Önce bir interface yazalım:**

**public interface ICar**

**{**

**void Drive();**

**void Stop();**

**}**

**Bu ne demek?**

* **Bir araba sınıfı yapacaksan,**
* **Drive (sürmek) ve Stop (durmak) fonksiyonları zorunlu olacak!**

**Ama *nasıl* sürecek, *nasıl* duracak, onu interface söylemiyor.  
Onu sen, arabayı yazarken belirliyorsun.**

**Şimdi bu interface'i bir sınıfta uygularsak:**

**public class Tesla : ICar**

**{**

**public void Drive()**

**{**

**Console.WriteLine("Tesla elektrikle sürüyor...");**

**}**

**public void Stop()**

**{**

**Console.WriteLine("Tesla otomatik fren yapıyor...");**

**}**

**}**

**Burada Tesla diye bir araba yaptık,  
ICar'ı uyguladığı için Drive ve Stop fonksiyonlarını yazmak zorunda kaldık.**

**🔵 Neden Interface kullanıyoruz?**

* **Kuralları garantiye almak için  
  (Her araba mutlaka Drive ve Stop yapabilsin istiyoruz mesela.)**
* **Esnek kod yazmak için  
  (Bir sürü farklı araba yaparsın, hepsi farklı çalışır ama hepsi ICar olur.)**
* **Bağımlılığı azaltmak için  
  (Kodun bir kısmı sadece ICar'ı tanır, gerçekte Tesla mı BMW mi bilmez. Böylece sistem esnek olur.)**
* **Dependency Injection gibi ileri seviye sistemlerde çok işe yarıyor.**

**🔵 Override Ne Demek?**

**Override kelime olarak "üstüne yazmak, değiştirmek" demek.**

**C#’ta bir sınıfta bir fonksiyonun temel (base) sınıfta yazılmış halini,  
kendi istediğin gibi yeniden yazmana (üstüne geçmene)  
override denir.**

**🔵 Basit örnekle:**

**Diyelim bir base class'ın var:**

**public class Animal**

**{**

**public virtual void Speak()**

**{**

**Console.WriteLine("Hayvan bir ses çıkarıyor.");**

**}**

**}**

**Burada Speak() fonksiyonu virtual olmuş.  
Bu, "İsteyen alt sınıflar bunu değiştirebilir" demek.**

**Şimdi Animal'dan türeyen bir sınıf yazalım:**

**public class Dog : Animal**

**{**

**public override void Speak()**

**{**

**Console.WriteLine("Köpek havlıyor!");**

**}**

**}**

**Ne yaptık?**

* **Dog sınıfı Animal'dan miras aldı (: Animal).**
* **Speak() fonksiyonunu override ettik.**
* **Artık Dog için Speak() çalışınca "Hayvan bir ses çıkarıyor" değil, "Köpek havlıyor" diyecek.**

**🔵 Neden override yapıyoruz?**

* **Çünkü her hayvanın sesi farklı!**
* **Temel sınıfta bir genel hareket yazıyoruz,**
* **Alt sınıflarda bunu özelleştiriyoruz.**

**🔵 Şartlar**

* **Base class'ta metodun başına virtual yazmak zorundasın.**
* **Child class'ta (alt sınıfta) override kullanarak değiştiriyorsun.**

**Virtual → Override**

**// Tüm arabaları getir**

**var allCars = carRepository.GetAll();**

**// Sadece 2020 model arabaları getir**

**var cars2020 = carRepository.GetAll(car => car.Year == 2020);**

**// Sadece Ford markalı arabaları getir**

**var fordCars = carRepository.GetAll(car => car.Brand == "Ford");**

**// 2020 model ve Ford markalı arabaları getir**

**var fordCars2020 = carRepository.GetAll(car => car.Year == 2020 && car.Brand == "Ford");**

**Burada:**

**car => car.Year == 2020 bir lambda expression'ıdır.**

**Bu ifade car üzerinde bir filtreleme yapıyor: sadece 2020 model arabaları döndürüyor.**

**🟠 5. "Repositories katmanında her entity için repository oluşturun"**

**Repository** ne demek?  
Bir sınıfın (örneğin User, Instructor, Applicant) veritabanıyla iletişimini sağlayan özel bir **ara katmandır**.

Normalde her Entity için (User, Instructor vs.) tek tek CRUD işlemleri (Add, Update, Delete, Get) yazmak çok zahmetli olurdu.

Mesela:  
IUserRepository, IInstructorRepository, IApplicantRepository, IEmployeeRepository gibi interface'ler açıyorsun.

Bunlar şöyle görünür:

csharp

KopyalaDüzenle

public interface IUserRepository : IRepository<User>

{

// Şu anda ekstra bir şey eklemene gerek yok.

}

csharp

KopyalaDüzenle

public interface IInstructorRepository : IRepository<Instructor>

{

}

**Yani:**  
Her entity kendi repository'sine sahip olacak ama içeriklerini tek tek yazmak yerine **IRepository**'den kalıtım alacak.

**🟠 6. Core Katmanında Generic Repository Yapısı**

Şimdi burada bir **genel repository** oluşturuyorsun, yani bir kere yazıp tüm entity'ler için kullanıyorsun.  
Bu interface şudur:

public interface IRepository<T> where T : class

{

void Add(T entity);

void Delete(T entity);

void Update(T entity);

T GetById(int id);

List<T> GetAll();

}

Bu kodun anlamı şu:

* T bir entity tipi olacak. (User, Instructor vs.)
* Ona göre CRUD işlemleri otomatik olarak tanımlanacak.

Bu sayede 50 farklı entity olsa bile **her biri için ayrı ayrı Add, Update yazmana gerek yok**. Hepsi IRepository'yi kullanacak.

**🟠 7. Asenkron işlemler için IAsyncRepository ve EfRepositoryBase**

Şu anda yazdığımız repository, işlemleri **senkron** yapıyor.  
Yani: "Veri tabanına ekle, işlemin bitmesini bekle, sonra diğer işe geç" mantığında.

Ama **modern sistemlerde** genelde **asenkron** çalışılır.  
Yani: "İşlemi başlat, işlemin bitmesini beklemeden başka işlere geç" (performansı artırır).

Bu yüzden bir de **IAsyncRepository** yazıyoruz:

public interface IAsyncRepository<T> where T : class

{

Task<T> GetByIdAsync(int id);

Task<List<T>> GetAllAsync();

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(T entity);

}

Buradaki her şey **Task** döner. Task demek: "Bu işlem biraz zaman alacak, beklemeye gerek yok" demektir.

**Sonra** bunun gerçek çalışır halini bir sınıfa yazıyorsun: EfRepositoryBase.

csharp

KopyalaDüzenle

public class EfRepositoryBase<T> : IAsyncRepository<T> where T : class

{

protected readonly DbContext \_context;

public EfRepositoryBase(DbContext context)

{

\_context = context;

}

public async Task<T> GetByIdAsync(int id)

{

return await \_context.Set<T>().FindAsync(id);

}

public async Task<List<T>> GetAllAsync()

{

return await \_context.Set<T>().ToListAsync();

}

public async Task AddAsync(T entity)

{

await \_context.Set<T>().AddAsync(entity);

}

public async Task UpdateAsync(T entity)

{

\_context.Set<T>().Update(entity);

}

public async Task DeleteAsync(T entity)

{

\_context.Set<T>().Remove(entity);

}

}

Burada:

* DbContext: Veritabanı bağlantısı sağlar.
* Set<T>(): Entity tipine göre doğru tabloyu seçer.

**Özetle:**  
Asenkron repository yazınca, programın performansı artar. "Kullanıcı beklemesin" mantığı olur.

**🟠 8. Request-Response Pattern Nedir?**

Bu son madde çok teorik. Çok basitleştirerek anlatıyorum:

* **Request**: İstemci tarafından gelen veri paketidir.  
  Örnek: Kullanıcı sisteme kayıt olmak istiyor → RegisterUserRequest.
* **Response**: Sunucu tarafından dönen cevaptır.  
  Örnek: Kayıt başarılı mı, kullanıcı ID’si ne oldu? → RegisterUserResponse.

**Neden bu sistem var?** Çünkü dışarıdan gelen veriyi, içerideki sistemden **ayırmak** için çok temiz bir yoldur.

**🔥 Özetle**

* Her entity için repository var.
* Generic Repository yazıp iş kolaylaşıyor.
* Asenkron repository ile işlemler daha hızlı oluyor.
* Request-Response ile kodlar düzenli oluyor.