**ASP.NET HAKKINDA**



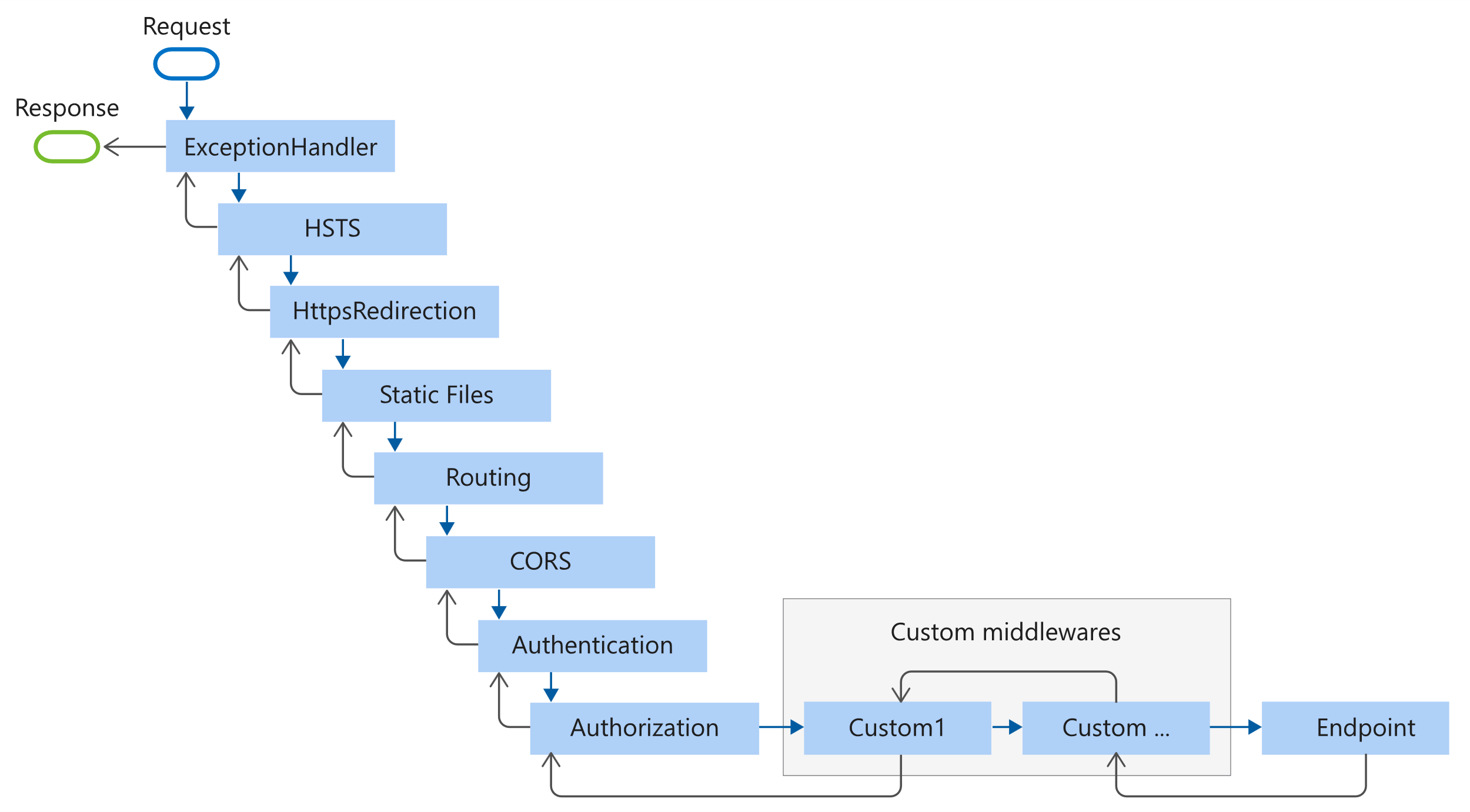
1. **PIPELINE VE MIDDLEWARE KAVRAMLARI**
2. **ASP.NET STRUCTURE**
3. **BUILD ve RUN**
4. **END POINT YAPISI**

ASP.NET, Microsoft tarafından geliştirilen bir web uygulama framework'üdür. Temel olarak web tabanlı uygulamalar geliştirmek için kullanılır ve birçok farklı platformda (Windows, macOS, Linux) çalışabilir. ASP.NET'in ana işlevleri:

1. **Web Uygulama Geliştirme:** ASP.NET, dinamik ve interaktif web siteleri ve uygulamaları geliştirmek için kullanılır. Sunucu taraflı işlemleri destekleyerek, kullanıcıların web tarayıcıları aracılığıyla etkileşimde bulunabilecekleri zengin içerikli web sayfaları oluşturmanıza olanak tanır.
2. **MVC ve API Desteği:** ASP.NET, Model-View-Controller (MVC) mimarisi ve API'lerin (Application Programming Interface) oluşturulması için güçlü bir altyapı sunar. MVC, web uygulamalarının veri, kullanıcı arayüzü ve iş mantığını ayrı bileşenler halinde organize etmenize yardımcı olur.
3. **Yüksek Performans:** ASP.NET, hızlı çalışma ve ölçeklenebilirlik için optimize edilmiştir. Bu özellikleri sayesinde büyük ölçekli uygulamaları destekleyebilir ve yüksek trafikli sitelerde performans sorunu yaşamadan çalışabilir.

ASP.NET'in farklı versiyonları bulunmakta olup, .NET platformunun güncel gelişmeleri ve teknolojik trendlere göre sürekli olarak güncellenmektedir.

**PIPELINE VE MIDDLEWARE KAVRAMLARI:**



Şekil 1: Pipeline

ASP.NET Core'un istek işleme pipeline'ı, bir dizi middleware bileşeninden oluşur. Bu pipeline, uygulamaya gelen her HTTP isteğini sırayla bu bileşenler aracılığıyla yönlendirir. Her middleware, isteği alır veya doğrudan bir yanıt üretir ya da bir sonraki middleware bileşenine isteği aktarır. Bu süreç, bir middleware yanıtı döndürene veya tüm bileşenler işletilene kadar devam eder. ASP.NET'de middleware, HTTP isteklerini işlemek ve yönlendirmek için kullanılan yazılım bileşenleridir. Middleware'ler, güvenlik, yönlendirme, hata yönetimi, API yönetimi ve kullanıcı oturumlarını yönetme gibi çeşitli işlevleri yerine getirebilir.

**ASP.NET STRUCTURE:**

metin, ekran görüntüsü, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

1. **Web UI**
   * **MVC (Model-View-Controller)**: Web uygulamaları için bir tasarım deseni sağlar. Kullanıcı arayüzü (view), veri modeli (model) ve uygulama mantığı (controller) arasında ayrım yapılmasına olanak tanır.
   * **Razor Pages**: MVC'ye benzer, ancak daha sayfa odaklı ve daha basit bir model-controller yaklaşımı sunar. Her Razor Page, kendi modeliyle birlikte gelir.
   * **SPA (Single Page Applications)**: Uygulamanın sayfalar arasında geçiş yaparken sayfayı yeniden yüklememesini sağlayan bir web uygulaması modelidir.
   * **Blazor**: WebAssembly kullanarak tarayıcıda C# kodu çalıştırma imkanı sunar, böylece JavaScript'e olan bağımlılığı azaltır.
2. **Services**
   * **HTTP APIs**: RESTful API'lar oluşturmak için kullanılır. HTTP protokolü üzerinden veri alışverişi yapar.
   * **SignalR**: Gerçek zamanlı web işlevselliği sağlar. Web sunucusu ve istemciler arasında anlık veri aktarımı yapılmasını mümkün kılar.
   * **gRPC**: HTTP/2 üzerinden çalışan, yüksek performanslı RPC (Remote Procedure Call) çerçevesidir. Google tarafından geliştirilmiş olup uzaktaki işlevleri lokalde kullanmaya yardımcı olur. Örneğin, bir e-ticaret platformunda ödeme işlemlerini yöneten bir mikro servis, kullanıcı verilerini güvenli bir şekilde işleyen başka bir mikro servisle iletişim kurmak için gRPC kullanabilir. Ödeme servisi, kullanıcı kimlik doğrulaması için kullanıcı servisine istek gönderir ve bu servisler arası etkileşim güvenli, hızlı ve verimli bir şekilde gerçekleşir. Bu süreç, platformun genel performansını ve kullanıcı deneyimini iyileştirir.
3. **Extensions**
   * **Logging**: Uygulama içindeki olayları kaydetme işlevselliği sağlar.
   * **Configuration**: Uygulama yapılandırmalarını yönetir.
   * **Options**: Yapılandırma ayarlarını güçlendirir ve uygulama içinde kolayca erişilebilir hale getirir.
   * **File Providers**: Dosya sistemiyle etkileşim için soyutlamalar sunar.
   * **Dependency Injection**: Bağımlılıkların yönetimi için bir teknik sunar ve sınıflar arası bağımlılıkları azaltır.
4. **Servers** 
   * **Kestrel**: Hafif, çok platformlu ve yüksek performanslı bir web sunucusudur.
   * **IIS (Internet Information Services)**: Microsoft'un güçlü bir web sunucusu ve uygulama sunucusudur.
   * **HTTP.sys**: Windows tabanlı uygulamalar için daha düşük seviyeli bir web sunucusudur.
5. **Middleware**
   * Alt kısımda listelenen bileşenler arasında Routing, Security, Localization, Caching, Compression, Session, Health Checks gibi çeşitli ara katman (middleware) işlevleri bulunur. Bu işlevler, gelen istekleri işlerken veya yönlendirirken ek işlevler sağlar.

Bu bileşenler, bir ASP.NET Core uygulamasının esnek ve modüler bir şekilde inşa edilmesini sağlar. Her bileşen, belirli bir ihtiyaca hizmet eder ve geliştiricilere, uygulamalarını ihtiyaçlarına göre özelleştirme yeteneği verir. Her biri kullanılmak zorunda olmayıp yukardaki tablodan da daha fazlası vardır.

**ASP.NET TASARIM DESENLERİ:**

.NET proje ve dosya şablonları, belirli türde uygulama veya dosya yapıları oluşturmak için önceden tanımlanmış kod ve yapılandırma dosyaları sağlar. Bu şablonlar, yazılım geliştirme sürecinde sık kullanılan proje türleri ve dosya yapılarını hızlı ve kolay bir şekilde başlatmak için kullanılır.(Yazılım mimarisi)

Proje şablonları, yeni bir proje oluştururken kullanılır ve belirli bir uygulama türüne yönelik tüm gerekli dosyaları ve yapılandırmaları içerir.

**Console Application**: dotnet new console -n MyConsoleApp

**Web Application:** dotnet new webapp -n MyWebAp

**Wep API**: dotnet new webapi -n MyApi

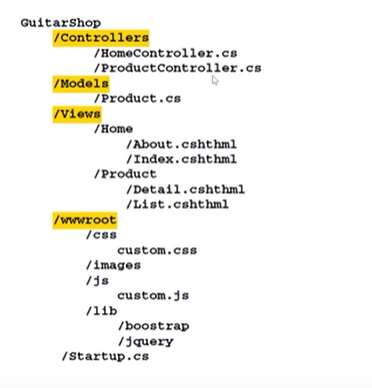
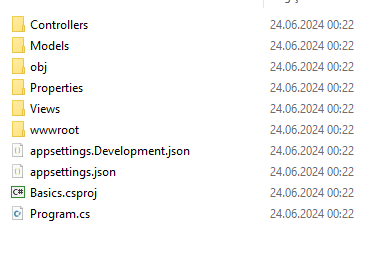
**MVC :** dotnet new MVC -o Basics

.

.

.

Web projeleri geliştirmede özellikle MVC şablonları ve türevleri çok kullanılır. Bu şablonun yapısını anlayacak olursak:



**Controllers**:

* **Amacı**: Uygulamanızın kontrol mantığını içeren dosyalar burada bulunur. Kontrolcüler, kullanıcı isteklerini alır ve uygun yanıtları döner.
* **Örnek**: HomeController.cs

**Models**:

* **Amacı**: Uygulamanızın veri yapılarını ve iş mantığını tanımlayan dosyalar burada bulunur. Modeller, veri tabanı ile uygulamanız arasındaki veri transferini sağlar.
* **Örnek**: Product.cs

**Views**:

* **Amacı**: Kullanıcıya gösterilecek HTML arayüzleri içeren dosyalar burada bulunur. Görünümler, kontrolcülerden gelen verileri kullanıcıya sunar.
* **Örnek**: Home/Index.cshtml

**wwwroot**:

* **Amacı**: Uygulamanızın statik dosyalarını (CSS, JavaScript, resimler vb.) barındırır. Bu dosyalar doğrudan kullanıcıya sunulabilir.
* **Örnek**: css/site.css

**obj**:

* **Amacı**: Projenin derleme işlemleri sırasında kullanılan geçici dosyaları içerir. Bu klasör, derleme ve yayınlama süreçleri için gereklidir.
* **Örnek**: project.assets.json

**Properties**:

* **Amacı**: Proje yapılandırma dosyalarını içerir. Proje ayarlarını ve özelliklerini belirler.
* **Örnek**: launchSettings.json

EMPTY TEMPLATE:

ASP.NET için "Empty Project Template" (Boş Proje Şablonu), geliştirme sürecine minimum başlangıç yapılandırması ile başlamak isteyen geliştiriciler için sunulan bir şablondur. Bu şablon, en temel yapılandırmayı sağlar ve geliştiricilere kendi uygulamalarını tamamen sıfırdan oluşturma esnekliği tanır.

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

* MVC (Model-View-Controller) veya Web API gibi belirli bir yapılandırma ile başlamaz. Bu nedenle, geliştirici bu yapılandırmaları manuel olarak eklemek zorundadır.
* Geliştiriciler, kendi ihtiyaçlarına göre uygulamanın mimarisini ve yapılandırmasını oluşturabilirler.
* Kendi bağımlılıklarını, ara yazılımları (middleware) ve servislerini manuel olarak ekleyebilirler.

Not: Program.cs dosyasına eklenecek middlerwareler genelde “Use..” ifadesi ile başlar

**BUILD ve RUN AŞAMASI:**

Tasarımının derlenmesi sırasında dotnet build komutu kullanılır. Bu komut, proje dosyalarını derler ve çalıştırılabilir hale getirir. İşlem sonucunda bir dizi dosya oluşturulur ve bu dosyalar projenin çıktısını temsil eder.

dotnet build komutu, .NET Core veya .NET projelerini derlemek için kullanılır. Bu komut, kaynak kodu alır ve çalıştırılabilir bir uygulama veya kitaplık oluşturur.

**Derleme Sonucunda Oluşan Dosyalar**

1. **DLL Dosyası (Dinamik Bağlantı Kitaplığı)**:
   * **Açıklama**: Derlenen kodun bulunduğu ana dosya. Proje adınıza göre adlandırılır (örneğin, MyWebApp.dll).
   * **Kullanım**: Bu dosya, uygulamanızın ana yürütülebilir kodunu içerir ve .NET çalışma zamanı tarafından yüklenir ve çalıştırılır.
2. **PDB Dosyaları (Program Veritabanı)**:
   * **Açıklama**: Hata ayıklama (debugging) bilgilerini içeren dosyalar. Proje adınıza göre adlandırılır (örneğin, MyWebApp.pdb).
   * **Kullanım**: Uygulamanızda hata ayıklama yaparken, bu dosyalar hata ayıklama araçlarının hangi satırda hata olduğunu anlamasına yardımcı olur.
3. **Config Dosyaları**:
   * **Açıklama**: Uygulamanızın yapılandırma ayarlarını içeren dosyalar (örneğin, appsettings.json).
   * **Kullanım**: Uygulamanın çalışma zamanı sırasında ihtiyaç duyduğu ayarları ve yapılandırmaları içerir.
4. **Değişiklikler (Dependencies)**:
   * **Açıklama**: Uygulamanızın çalışması için gerekli olan diğer kitaplıklar ve bağımlılıklar.
   * **Kullanım**: Bu dosyalar, uygulamanızın doğru şekilde çalışması için gerekli olan tüm bağımlılıkları içerir. Bu bağımlılıklar genellikle bin ve obj klasörlerinde bulunur.
5. **EXE Dosyası (Yürütülebilir Dosya)**:
   * **Açıklama**: Bazı .NET projelerinde (özellikle konsol uygulamaları için) oluşturulabilir.
   * **Kullanım**: Uygulamanızı çalıştırmak için doğrudan kullanılabilir. ASP.NET Core MVC projelerinde genellikle DLL dosyası kullanılır, ancak dotnet run komutu ile çalıştırılır.

**Derleme Sonucu Ortaya Çıkan Yapı**

Derleme işlemi sonucunda proje klasörünüzde **bin** ve **obj** adında iki ana klasör oluşur:

* **bin**: Derlenen uygulamanızın çıkış dosyalarını içerir. Örneğin, *bin/Debug/netcoreapp3.1* veya *bin/Release/netcoreapp3.1* gibi. .NET projelerinde derleme (build) işlemi sırasında oluşturulan ve derlenen çıktı dosyalarının yer aldığı bir klasördür. Bu klasör, uygulamanızın çalıştırılabilir ve bağımlılık dosyalarını içerir.
* **obj**: Geçici derleme dosyalarını ve proje varlıklarını içerir.

**Uygulama Çalıştırma**

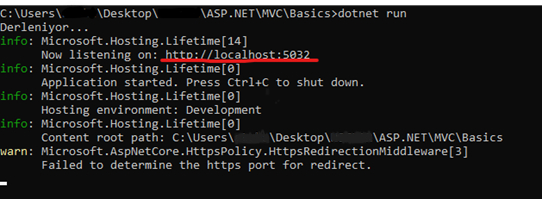
Derlenen uygulamanızı çalıştırmak için aşağıdaki komutu kullanabilirsiniz:

**dotnet run**

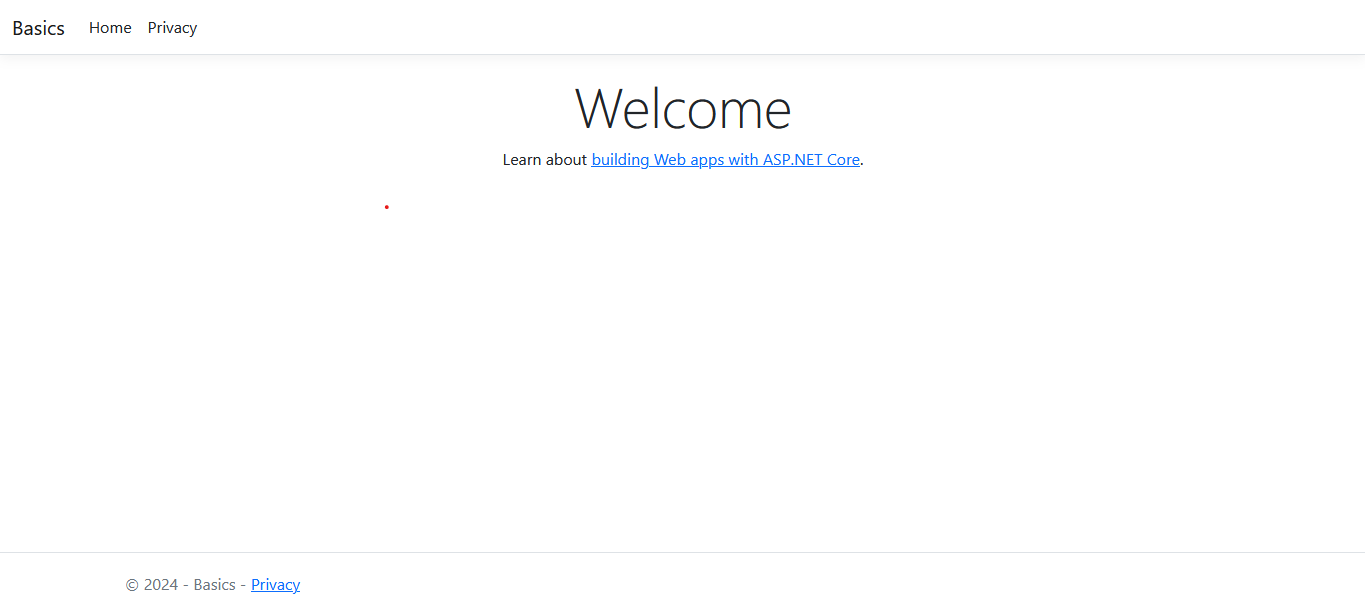
Veya doğrudan DLL dosyasını çalıştırabilirsiniz:

dotnet MyWebApp.dll

Bu komutlar, uygulamanızı çalıştırır ve çalışabilir bir durumda sunar. ("dotnet watch” komutu “dotnet run” komutuna ek olarak sunucuyu sürekli olarak izler ve projedeki küçük değişikliklerde anlık olarak günceller. Bu sayede sunucuyu her değişiklikte kapatıp açmaya gerek kalmadan sayfayı güncelleyerek değişiklikleri görürüz.)



http://localthost:5032 ‘ye gidilirse aşağıdaki sayfa görülür:



**ENDPOINT YAPISI:**

ASP.NET Core'da endpoint yapısı, web uygulamanızın farklı URL'lerine nasıl yanıt vereceğini tanımlayan yapıdır. Endpoint'ler, genellikle bir Startup.cs dosyasında veya ASP.NET Core 3.0 ve sonrasında daha yaygın olarak kullanılan **Program.cs** dosyasında yapılandırılır. (Endpoint bir url olarak düşünülebilir. Örneğin ürünler için https://mysites/product şeklinde bir endpoint olabilir.)

metin, yazı tipi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

Yukarıdaki program.cs dosyasındaki app.MapControllerRoute()’dan anlaşılacağı gibi bir uç noktanın yapısı /controller/action/id değişkenlerini içerek şekilde olmalıdır. Bu değişkenle varsayılan olarak sırasıyla Home/Index değerlerini almıştır.

/, /Home , /Home/Index her üç durumda da aynı endpointe yönlendirme yapılır. Örneğin, yukarıdaki [http://localhost:5032](http://localhost:5032/) için default değerler aslında kök dizini olup <http://localhost:5032/>, <http://localhost:5032/Home>, <http://localhost:5032/Home/Index> hepsi birbiri ile aynıdır.

**Controller Adı Çözümlemesi**: URL'deki controller ismi {controller} segmentiyle eşleştirilir. Örneğin, /Course URL'si için MVC, Course kelimesini alır ve ardına Controller ekleyerek CourseController sınıfını arar. Bu, MVC'nin controller isimlendirme kuralına dayanır: Controller sınıflarının adı, URL'deki isimlerinin sonuna "Controller" kelimesinin eklenmesiyle oluşur. Yani MVC modeli içinde Controllers klasörü içinde CourseController sınıfı aranır ve ona göre sonuç döndürülür. Eşleşen controller sınıfı yoksa, hata mesajı gösterilir. Bulunursa, controller sınıfı bir örnek olarak oluşturulur.

EK:

ASP.NET Core MVC yönlendirme sistemi URL'lerde büyük/küçük harfe duyarlı değildir. Örneğin:

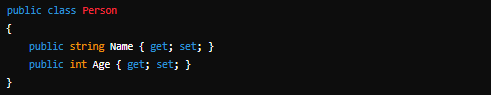
* /Home/Index
* /home/index
* /HOME/INDEX

Yukarıdaki URL'lerin tümü aynı controller ve action'a yönlendirilir.

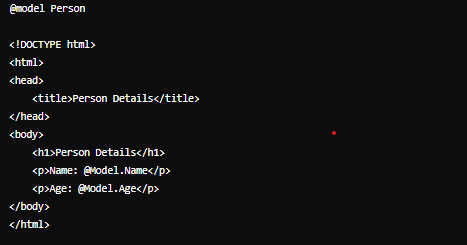
**MODEL DİREKTİFİ:**

@model direktifi, bir ASP.NET Core Razor sayfasında (CSHTML dosyasında) kullanılır ve bu sayfanın modeliyle ilgili bilgi sağlar. Bu direktif, Razor sayfasının belirli bir veri modeline (genellikle bir C# sınıfı) bağlanmasını sağlar, böylece bu modelin özellikleri ve yöntemleri Razor sayfasında doğrudan kullanılabilir.

**Razor sayfası, ASP.NET Core içinde kullanılan bir şablon motorudur ve web uygulamaları geliştirmek için kullanılır**. Razor, HTML ve C# kodlarını birleştirmenizi sağlayan bir syntax (sözdizimi) sunar. Razor sayfaları, ".cshtml" dosya uzantısına sahiptir ve dinamik web içeriklerini oluşturmak için kullanılır. Özetle, HTML ve C# kodlarını aynı dosyada kullanmanıza izin verir, böylece dinamik web içerikleri oluşturmak daha kolay hale gelir.



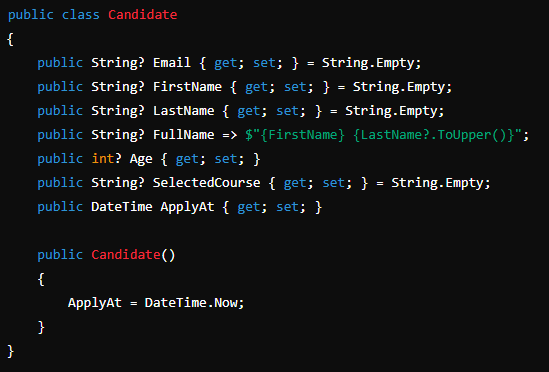
Şekil 2: csharp



Şekil 3: cshtml

Bu örnekte, Person sınıfı model olarak belirtilmiştir. Sayfa içinde @Model ifadesi ile bu modelin özelliklerine erişilir ve HTML içeriğinde görüntülenir. Yani “@Model” ifadesi “Person” classının bir temsilidir.

**C# Hakkında ve Temel Sözdizimi:**



**Yapıcı Metot (Constructor) Nedir?**

* **Yapıcı Metotlar**: Bir sınıfın örneği oluşturulduğunda otomatik olarak çağrılan özel metotlardır. Yapıcı metotlar, sınıfın adını taşır ve dönüş türü belirtilmez. Yapıcı metotlar, nesne başlatma işlemlerini gerçekleştirir.

Yukarıdaki örnekte, Candidate sınıfının yapıcı metotu Candidate() şu işlemleri yapar:

1. **Başlatma**: Candidate sınıfının bir örneği oluşturulduğunda çağrılır.
2. **ApplyAt Ayarı**: Yapıcı metot içinde, ApplyAt özelliği mevcut tarih ve saat ile başlatılır (DateTime.Now).

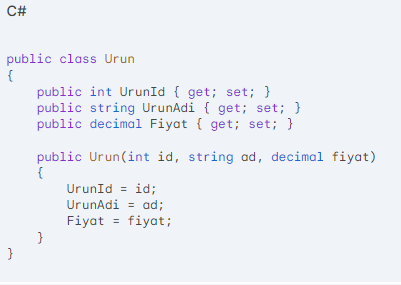
C# dilinde değişkenler, veri tipleri ile birlikte tanımlanır ve bu değişkenler sınıf içinde özellik (property) olarak adlandırılır. Örneğin:

* public String? Email { get; set; } = String.Empty;

Bu, Email adında bir özelliği (property) tanımlar. public erişim belirteci, bu özelliğin sınıfın dışından erişilebilir olduğunu belirtir. String? türü, bu özelliğin bir String türünde olduğunu ve nullable (boş olabilir) olabileceğini belirtir. get; set; ise bu özelliğin hem okunabilir hem de yazılabilir olduğunu ifade eder. =String.Empty, Email özelliğinin varsayılan değerinin boş bir string olduğunu belirtir. Aynı mantık yukarıdaki diğer özellikler için de geçerlidir.

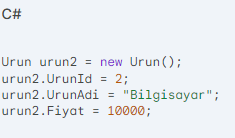
KULLANICI TANIMLI BİR SINIFTAN NESNE OLUŞTURMA VE FARKLI DEĞER ATAMA YÖNTEMLERİ:

1)

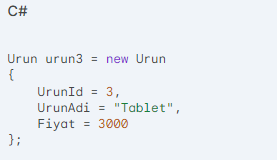


Urun urun1 = new Urun(1, "Telefon", 5000);

2)



3)



**Urun urun1 = new Urun(1,”Telefon”,5000);** satırındaki nesnenin veri tipi, **Urun** sınıfıdır.

**PRG PATTERN NEDİR:**

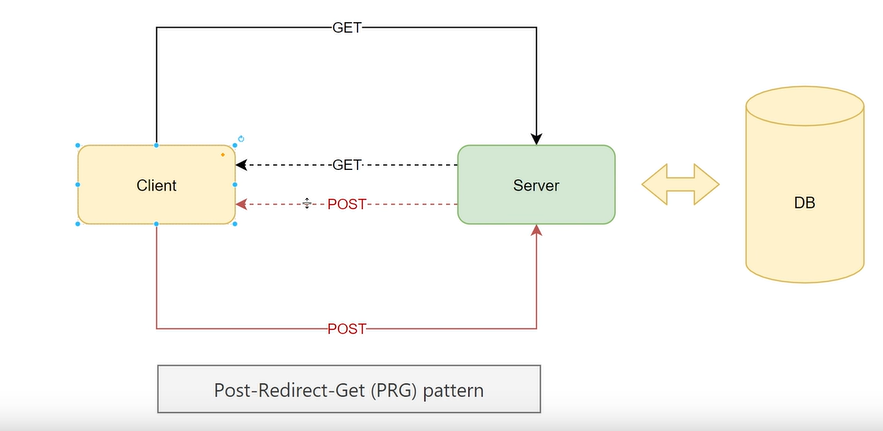
PRG (Post/Redirect/Get) pattern, web uygulamalarında yaygın olarak kullanılan bir tasarım desenidir. Bu desen, form işlemlerinin kullanıcı deneyimini iyileştirmek ve tekrarlanan form gönderimlerini önlemek için kullanılır. PRG deseni, özellikle form gönderimlerinde "Yeniden Gönder" sorununu (postback problem) çözmek için etkili bir yöntemdir.

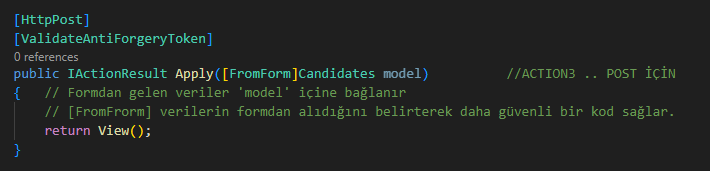
**PRG Pattern Adımları**

1. **Post**: Kullanıcı bir formu doldurur ve gönderir. Bu işlem, bir POST isteği başlatır ve veriler sunucuya gönderilir.
2. **Redirect**: Sunucu, POST isteğini işler ve verileri aldıktan sonra, kullanıcıyı bir GET isteği ile başka bir sayfaya yönlendirir. Bu yönlendirme genellikle bir HTTP 302 (Found) yanıtı ile yapılır.
3. **Get**: Tarayıcı, belirtilen yeni URL'ye GET isteği gönderir ve kullanıcıya sonuç sayfasını gösterir. Bu sayfa, formun başarıyla işlendiğini ve sonucun görüntülendiğini belirtir.

**PRG Pattern'in Avantajları**

* **Yeniden Gönderme Sorununu Önler**: Kullanıcı, sayfayı yenilediğinde form tekrar gönderilmez. Bu, istemeden çift form gönderimlerini ve veri işleme hatalarını önler.
* **Kullanıcı Deneyimini İyileştirir**: Kullanıcı, başarılı bir işlemden sonra sonuç sayfasına yönlendirilir, bu da daha anlaşılır ve kullanıcı dostu bir deneyim sağlar.
* **SEO Dostu**: GET isteği ile sonuç sayfası erişilebilir hale gelir, bu da arama motorları tarafından dizine eklenebilir.



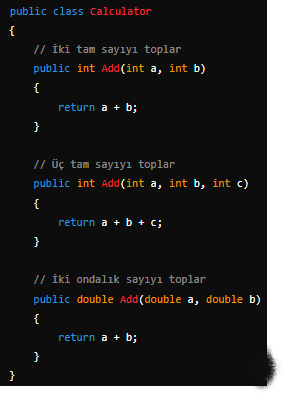


Yukarıdaki kod parçası kullanıcı **Model Binding’e** örnektir. Buna göre kullanıcının formda doldurduğu veriler modele bağlanacaktır. Candidate, formdan gönderilen verileri temsil eden bir sınıftır. ASP.NET Core, form verilerini bu modele otomatik olarak bağlar.

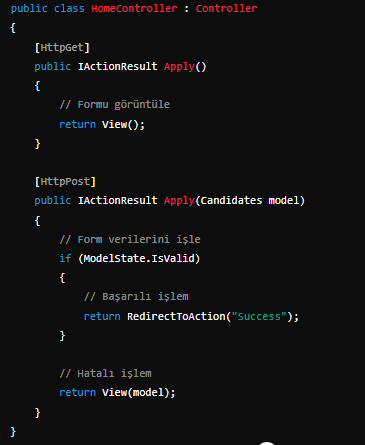
Yukarıda fark edileceği üzere aynı isimli iki ayrı Apply fonksiyonu vardır. Bu nasıl olur?

**Metod aşırı yükleme (method overloading)**, nesne yönelimli programlamada (OOP) sıkça kullanılan bir tekniktir ve bir sınıfın içinde aynı isimle birden fazla metod tanımlamayı mümkün kılar. Bu metodlar, parametre listelerinin (parametre sayısı, tipi veya sırası) farklı olması ile birbirinden ayırt edilir.

Metod aşırı yüklemenin OOP ile ilişkisi, **Polimorfizm’dir.** Metod aşırı yükleme, polimorfizmin bir biçimi olarak kabul edilir. Polimorfizm, bir nesnenin farklı şekillerde davranabilme yeteneğini ifade eder. Aşırı yükleme, aynı metod adının farklı parametrelerle çağrılarak farklı işler yapmasını sağlar.

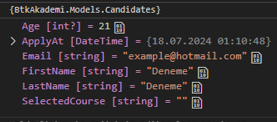


Yukarıdaki basit örnekte, Add metodunun üç farklı aşırı yüklenmiş versiyonu vardır. Her biri farklı parametre listelerine sahiptir.



Aynı şekilde bu örnekte de, Apply metodunun GET ve POST versiyonları vardır. Her biri farklı HTTP isteklerini işler ve farklı işlevler gerçekleştirir.(NOT: Metot üzerinde herhangi bir dekoratif yoksa varsayılan olarak GET’dir.)

\*NOT: ASP.NET Core MVC'de formdaki input elemanlarının modeldeki property'ler ile eşleşebilmesi için, input ***elemanlarının “name” attribute'u modeldeki property isimleriyle birebir aynı olmalıdır***. id attribute'u eşleşme için kullanılmaz; sadece HTML'de belirli bir input elemanını tanımlamak için kullanılır.



Yukarıdaki görselden de anlaşılacağı üzere Model Binding işlemi başarılıyla sağlanmıştır. Formdan gelen veriler yakalanmıştır.(Candidates modeli tarafından)

**C# ‘ da Fonksiyon Tanımlama:**

[erişim belirleyicisi] [geri dönüş tipi] [metod adı]([parametre listesi]) {

// Metodun gövdesi (kodlar) [return ifadesi];

}

1 **Erişim Belirleyicisi**: Bu, metodun erişim düzeyini belirler. En yaygın erişim belirleyiciler şunlardır:

* public: Metoda her yerden erişilebilir.
* private: Metoda sadece aynı sınıf içinden erişilebilir.
* protected: Metoda yalnızca aynı sınıf veya türetilmiş sınıflardan erişilebilir.
* internal: Metoda sadece aynı proje içerisindeki kodlardan erişilebilir.

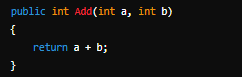
2 **Geri Dönüş Tipi**: Metodun döndüreceği veri türünü belirtir. Eğer metod herhangi bir değer döndürmeyecekse, void anahtar kelimesi kullanılır. Yapılanıdırıcı metodlarda geri dönüş tipi belirtilmez.

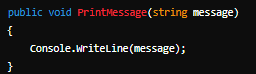
3 **Metod Adı**: Metodun adıdır ve onu tanımlayan isimdir.

4 **Parametre Listesi**: Metodun alacağı parametreleri belirtir. Parametreler, veri türleri ve isimleriyle birlikte parantez içinde tanımlanır. Parametre almayacaksa parantezler boş bırakılır.

5 **Metod Gövdesi**: {} içinde yer alan, metodun çalıştırılacak kodlarını içerir.

6 **return İfadesi**: Metodun sonucunu döndürmek için kullanılır. Eğer metod void döndürüyor (yani hiçbir şey döndürmüyor) ise return ifadesi gerekli değildir.





**TAG HELPERS**

Tag Helpers, ASP.NET Core freamworkünde bulunan bir özellik olup Razor sayfalarında ve Razor View'larında HTML etiketlerine sunucu tarafı işlevsellik ekleyerek birçok avantaj sağlar. HTML etiketlerini programlama dili ile genişletmenizi sağlar, Razor dosyalarında (örneğin, .cshtml dosyaları) HTML etiketlerine dinamik içerik veya işlevsellik eklemek için kullanılır.



Bu kullanım, asp-for özniteliği sayesinde UserName modeline bağlanarak, dinamik olarak bir input oluşturur.



Bu kullanım, asp-controller ve asp-action öznitelikleri ile dinamik olarak bir bağlantı oluşturur.

**REPOSITORY(In Memory):**

 **İş Mantığı (Business Logic):**

* İş mantığı, uygulamanın gerçekleştirdiği temel işlemler ve kurallar kümesidir. Örneğin, bir e-ticaret sitesinde sipariş oluşturma, ödeme işlemleri ve envanter yönetimi iş mantığıdır.
* İş mantığı, genellikle iş kurallarını, veri işleme süreçlerini ve kullanıcı girişleri gibi uygulamanın çekirdek fonksiyonlarını içerir.

 **Veri Erişim Detayları (Data Access):**

* Veri erişim detayları, uygulamanın veritabanı veya diğer veri depolarıyla nasıl etkileşime geçtiğini tanımlar. Bu, veritabanı sorguları, CRUD (Create, Read, Update, Delete) işlemleri ve veri depolama işlemlerini içerir.
* Veri erişim kodu genellikle SQL sorguları, Entity Framework veya Dapper gibi ORM araçları ile yazılır.

Veri tabanına erişim, veri ekleme, güncelleme, silme ve okuma işlemlerini yönetir. Repository, bu işlemleri gerçekleştirirken iş mantığını veri erişim detaylarından ayırır, böylece kodun daha temiz, modüler ve bakımı kolay olur.

MVC (Model-View-Controller) modelinde iş mantığı **Controller** kısmında gerçekleşirebilir**. Ancak, ideal bir yapıda iş mantığının büyük bir kısmı Model katmanında ya da ek bir ara katman olan özellikle de "Service" katmanları gibi daha spesifik ayrı katmanlarda gerçekleşir.**

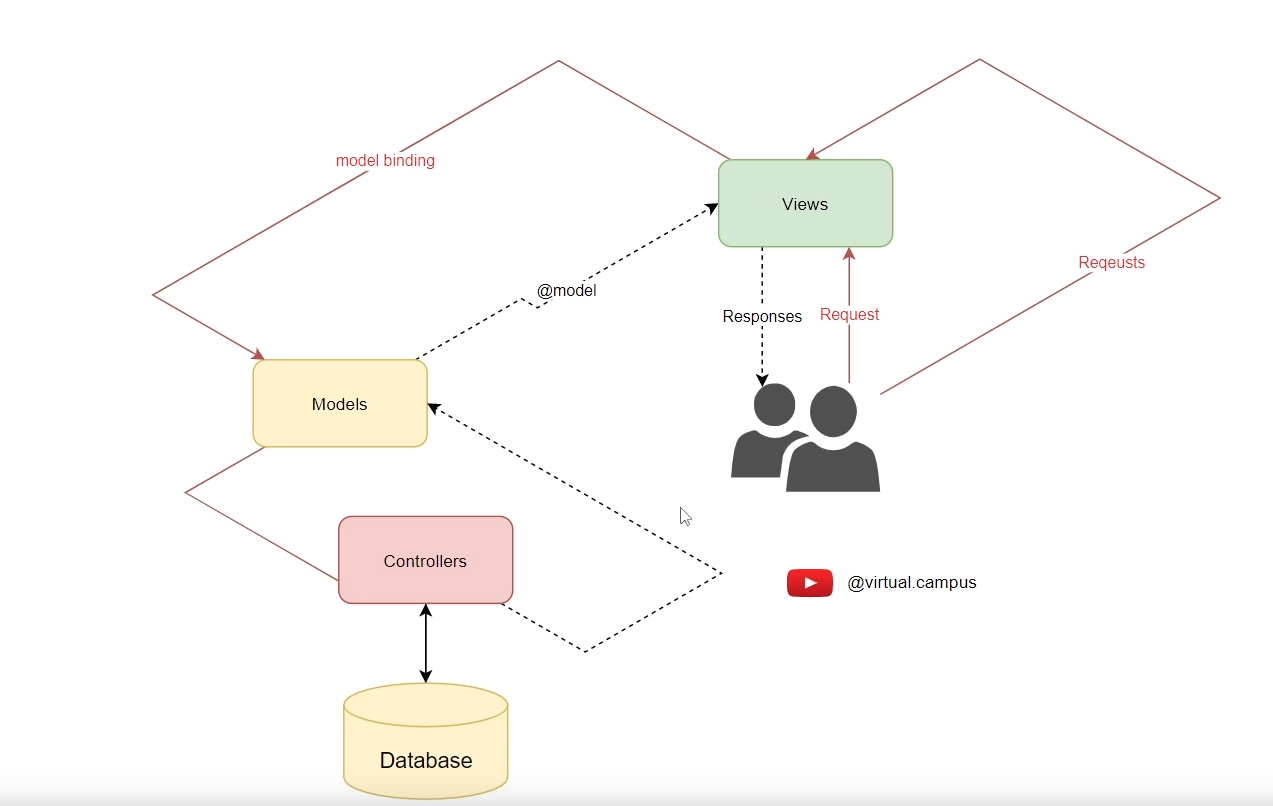
İdeal bir MVC yapısında***, Controller sadece gerekli model ve view arasında bir köprü görevi görür***. İş mantığının çoğu, modelin içinde veya daha özel olarak ayrı bir iş mantığı katmanında (örneğin, servis katmanı) yer alır. Bu yaklaşım, aşağıdaki avantajları sağlar:

* **Test Edilebilirlik:** İş mantığını ayrı bir katmanda toplamak, birim testleri yazmayı ve uygulama mantığını test etmeyi kolaylaştırır.
* **Bakım ve Yönetilebilirlik:** İş mantığı ve veri erişim kodlarının merkeziyetçi yönetimi, uygulamanın bakımını ve güncellemelerini kolaylaştırır.
* **Yeniden Kullanılabilirlik:** Aynı iş mantığını farklı uygulamalarda kullanmak mümkün hale gelir.

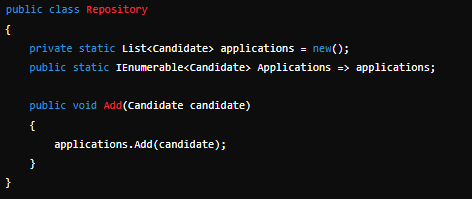
Sonuç olarak, MVC modelinde iş mantığının, genellikle model ve özellikle servis katmanlarında yoğunlaştırılması tercih edilirken, controller sadece bu işlevleri koordine etmekle sorumlu olur. **Servis katmanı**, genellikle controller ve model katmanları arasında yer alır ve iş kurallarını, veri işleme mantığını ve uygulama mantığını yönetir. Bu katman, uygulamanın modülerlik ve yeniden kullanılabilirlik seviyesini artırarak, genişletilebilir ve bakımı kolay sistemler inşa etmeye olanak tanır.

metin, yazı tipi, sayı, numara, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu



ASP.NET'de bir in-memory repository yapısı oluşturmak, genellikle test amaçlı senaryolarda veya basit uygulamalarda kullanılan bir yöntemdir. Bu tür bir repository, verileri geçici olarak bellekte saklar ve genellikle bir veri tabanına ihtiyaç duymadan uygulama içinde veri işleme ve depolama işlemleri yapmak için kullanılır.



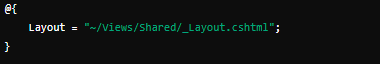
Bu basit kod parçasının işlevi:

* **private static List<Candidate>**: Bu, Candidate türünde elemanlar içeren bir List tanımlar. private erişim belirleyicisi, bu listenin yalnızca bu sınıf içinden erişilebilir olduğunu belirtir.
* **static**: Bu, listenin sınıf düzeyinde bir değişken olduğunu ve sınıfın tüm örnekleri arasında paylaşıldığını belirtir.
* **new List<Candidate>()**: Bu, List<Candidate> türünde yeni bir liste oluşturur. new() ifadesi, C# 9.0 ve sonrasında kullanılan bir kısayoldur ve new List<Candidate>() ile aynıdır. Örneği oluşturur.
* **public static IEnumerable<Candidate>**: Bu, Candidate türünde elemanlar içeren bir IEnumerable tanımlar. public erişim belirleyicisi, bu özelliğin sınıfın dışından erişilebilir olduğunu belirtir. static ise bu özelliğin sınıf düzeyinde olduğunu ve sınıfın tüm örnekleri arasında paylaşıldığını belirtir.
* **Applications => applications**: Bu, applications listesini IEnumerable olarak dışa açar. Bu, listenin yalnızca okunabilir olmasını sağlar, böylece dışarıdan listeye ekleme, silme veya güncelleme yapılamaz.
* **public void Add(Candidate candidate)**: Bu, Candidate türünde bir elemanı listeye eklemek için kullanılan herkese açık bir metottur.
* **applications.Add(candidate)**: Bu, parametre olarak verilen candidate nesnesini applications listesine ekler.

Bu In Memory bir yöntem olup veriler veri tabanı yerine bellek üzerinde geçici olarak test uygulamarı veya örnek uygulamalar için tutulur.

**BASEHTML:**

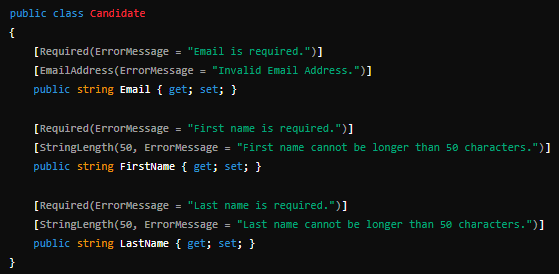
ViewStart.cshtml, ASP.NET MVC ve ASP.NET Core MVC projelerinde kullanılan özel bir Razor görünüm dosyasıdır. Bu dosya, uygulamanızdaki tüm Razor görünümleri için ortak ayarları ve yapılandırmaları tanımlamak için kullanılır.



Bu örnekte, ViewStart.cshtml dosyası, Layout dosyasını tüm Razor görünümleri için \_Layout.cshtml olarak ayarlar.

**MODEL VALIDATION:**

Model Validation (Model Doğrulama), bir modelin (veri sınıfının) belirli kurallara ve kısıtlamalara uygun olup olmadığını kontrol etme sürecidir. Bu doğrulama, veri girişlerinin belirli kriterlere uymasını sağlayarak uygulamanın güvenliğini ve sağlamlığını artırır**.**



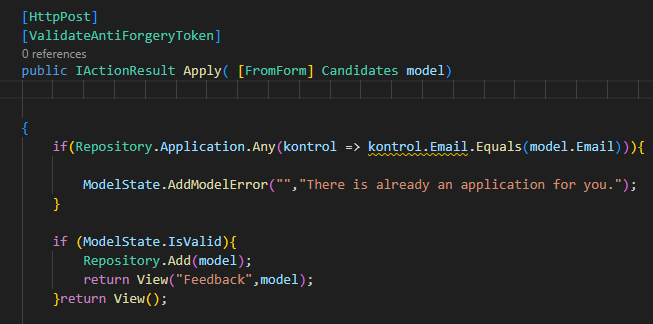
* [Required]: Özelliğin boş bırakılamayacağını belirtir.
* [EmailAddress]: Özelliğin geçerli bir e-posta adresi formatında olmasını sağlar.
* [StringLength]: Özelliğin maksimum uzunluğunu belirtir.

Bu hata mesajları <div class="text-danger" asp-validation-summary="All"></div> etiketi ile gösterilirir.

asp-validation-summary etiketini, formunuzda genel hata mesajlarını göstermek istediğiniz yere yerleştirirsiniz.(Yukarıdaki ErrorMessage=”..”)

ModelState doğrulaması sırasında oluşan tüm hata mesajları bu özet bölümünde gösterilir. ModelState, doğrulama sonuçlarını ve hata mesajlarını içerir. ModelState.IsValid özelliği, modelin doğrulama kurallarına uyup uymadığını kontrol eder. ModelState.IsValid, modelde tanımlanan tüm doğrulama kurallarına (örneğin, [Required], [EmailAddress], [StringLength], vb.) göre modelin geçerli olup olmadığını kontrol eder. Bu kurallardan herhangi biri ihlal edilirse, ModelState geçersiz(false) kabul edilir.

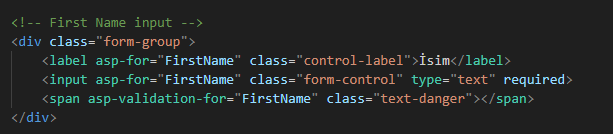




Repository.Application.Any(kontrol => kontrol.Email.Equals(model.Email)): Repository.Application koleksiyonunda modelin e-posta adresiyle eşleşen bir kayıt olup olmadığını kontrol eder.Eğer eşleşen bir kayıt bulunursa, ModelState.AddModelError("", "There is already an application for you."); ile model durumuna bir hata eklenir. Bu, form tekrar görüntülendiğinde kullanıcıya bir hata mesajı gösterecektir.

* if (ModelState.IsValid): Bu koşul, model durumunun geçerli olup olmadığını kontrol eder. ModelState.IsValid true ise, model tüm doğrulama kurallarını geçmiştir.
* Repository.Add(model): Model geçerli ise, model Repository'ye eklenir.
* return View("Feedback", model): Başarılı bir ekleme işlemi sonrası, "Feedback" adlı bir görünüm geri döndürülür ve modele ait veriler bu görünüme geçirilir.
* Geçerli değilse return View() ile sayfa geri döndürülür.

HTML5 required Etiketi Kullanımı:



HTML5'te required etiketi kullanılarak form alanlarının doğrulaması yapılabilir. Bu yöntem istemci tarafı doğrulaması sağlar ve kullanıcıların formu eksik bilgiyle göndermesini önlemek için tarayıcı tarafından kontrol edilir. Bu doğrulama türü, kullanıcı deneyimini iyileştirir çünkü form gönderilmeden önce eksik veya yanlış girilen bilgiler kullanıcıya anında bildirilir.

Ancak, yalnızca HTML doğrulamasına güvenmek güvenlik açısından yeterli değildir. Sunucu tarafı doğrulaması her zaman gereklidir çünkü istemci tarafı doğrulaması kolayca atlatılabilir. Sunucu tarafında yapılan doğrulama, verilerin gerçekten doğru ve güvenilir olduğunu garanti eder.

**NuGET:**

metin, ekran görüntüsü, yazı tipi, diyagram içeren bir resim

Açıklama otomatik olarak oluşturuldu

NuGet, .NET platformu için bir paket yöneticisidir. Geliştiricilere, .NET projeleri için gerekli olan kütüphaneleri ve araçları kolayca bulma, yükleme ve yönetme imkanı sunar. Microsoft tarafından geliştirilmiş olan NuGet, hem açık kaynaklı hem de kapalı kaynaklı paketlerin dağıtımını destekler.

**Paket yöneticisi**, yazılım projelerinde kullanılan kütüphane ve bağımlılıkları (dependencies) yönetmek için kullanılan bir araçtır. Paket yöneticileri, yazılım geliştirme sürecinde tekrar kullanılabilir kodların kolayca entegre edilmesini sağlar ve bu kodların (paketlerin) doğru sürümlerini ve bağımlılıklarını yönetir.

NuGet, projelere gerekli olan kütüphaneleri veya bileşenleri kolayca eklemenizi sağlar. Örneğin, bir proje için JSON işleme kütüphanesi (Newtonsoft.Json) gibi bir kütüphane gerekiyorsa, NuGet aracılığıyla bu kütüphaneyi kolayca ekleyebilirsiniz. ( <https://www.nuget.org/> )

Örneğin proje içinde aşağıdaki gibi bir komut çalıştırarak Sqlite paketini projeye ekleyebiliriz. Ekli paketleri görmek için dotnet list package komutu kullanılabilir.

dotnet add package Microsoft.EntityFrameworkCore.Sqlite.Core --version 8.0.0

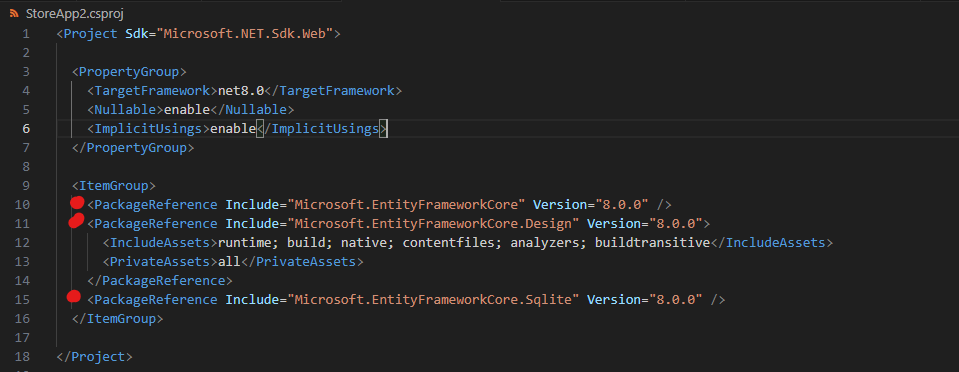
dotnet add package Microsoft.EntityFrameworkCore --version 8.0.0

dotnet add package Microsoft.EntityFrameworkCore.Design --version 8.0.0

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Aynı şekilde .csproj dosyası içinde paketlere referans verildiği doğrulanabilir.



**VERİ ODAKLI GELİŞTİRME:**

Veri odaklı geliştirme, veri tabanı yönetim sistemleriyle etkileşimde bulunan uygulamalar geliştirme sürecini ifade eder. Bu, veri tabanından veri okuma, yazma, güncelleme ve silme işlemlerini içerir. .NET ekosisteminde, bu süreçleri kolaylaştıran çeşitli araçlar ve çerçeveler bulunmaktadır. Entity Framework’e bakacak olursak;

**Entity Framework (EF)**

Entity Framework, .NET için bir Object-Relational Mapping (ORM) aracıdır. Veri tabanı ile çalışmayı kolaylaştırır, geliştiricilerin SQL yazmasına gerek kalmadan veri tabanı işlemlerini gerçekleştirmesine olanak tanır.

ORM’ye değinecek olursak;

ORM, yazılım geliştirme süreçlerinde kullanılan bir tekniktir ve veri tabanı sistemleri ile nesne yönelimli programlama dilleri arasında bir köprü kurar. ORM, veri tabanı tabloları ile programlama dilindeki sınıflar arasında otomatik olarak eşlemeler yaparak, veri tabanı işlemlerinin daha kolay ve yönetilebilir hale gelmesini sağlar. ORM'in temel amacı, SQL sorguları yazmadan veri tabanı işlemlerini gerçekleştirebilmektir.

ORM'nin başlıca özellikleri ve avantajları şunlardır:

1. **Nesne-Tablo Eşleme (Mapping):** ORM araçları, veri tabanı tablolarını programlama dilindeki sınıflara, tablolardaki satırları ise bu sınıfların nesnelerine dönüştürür. Bu sayede veri tabanı işlemleri, doğrudan nesneler üzerinden yapılabilir.
2. **Veri tabanı Bağımsızlığı:** ORM araçları, uygulamanın belirli bir veri tabanına bağımlı olmadan çalışmasını sağlar. Farklı veri tabanları arasında geçiş yapmak gerektiğinde, genellikle sadece ORM yapılandırmalarını değiştirerek bu işlem gerçekleştirilebilir.
3. **Kod Yazımını Azaltır:** ORM kullanımı, SQL sorguları yazma gereksinimini ortadan kaldırarak, daha az ve daha temiz kod yazılmasını sağlar. Bu durum, hem geliştirme sürecini hızlandırır hem de kodun bakımını kolaylaştırır.
4. **Güvenlik:** ORM araçları, SQL enjeksiyonu gibi güvenlik açıklarına karşı koruma sağlar. Hazır fonksiyonlar ve parametrik sorgular kullanılarak, veritabanı işlemleri daha güvenli hale getirilir.

Veri tabanı odaklı geliştirmede **.sln dosyası (solution file)** kullanmak, projelerinizi organize etmenin ve yönetmenin standart bir yoludur.

Bir .sln dosyası (solution file), Microsoft Visual Studio tarafından kullanılan bir çözüm dosyasıdır ve birden fazla proje dosyasını ve bunların bağımlılıklarını organize eden bir kapsayıcıdır. Çözüm dosyaları, projelerin ve bileşenlerin nasıl bir arada çalıştığını tanımlar ve Visual Studio gibi geliştirme ortamlarında projeleri yönetmeyi kolaylaştırır.

**.sln Dosyasının İşlevleri**

1. **Projelerin Organizasyonu**: Birden fazla projeyi bir arada tutar ve bu projelerin birbirleriyle olan ilişkilerini yönetir.
2. **Bağımlılık Yönetimi**: Projeler arasındaki bağımlılıkları tanımlar ve bu bağımlılıkların nasıl yapılandırılacağını belirler.
3. **Derleme ve Yapılandırma**: Projelerin nasıl derleneceği ve hangi yapılandırmaların (örneğin, Debug veya Release) kullanılacağı gibi bilgileri içerir.
4. **Visual Studio Entegrasyonu**: Visual Studio'ya projelerin nasıl açılacağını, hangi projelerin derleneceğini ve hangi dosyaların düzenleneceğini bildirir.

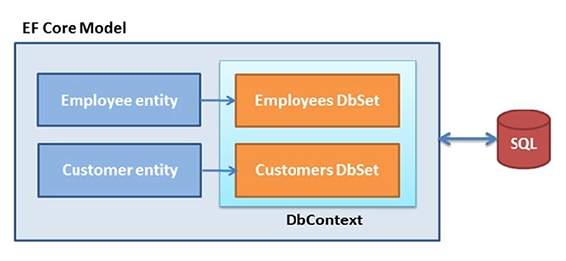




* Bu komut, MySolution.sln adında bir çözüm dosyası oluşturur.
* Bu komut, MySolutionDirectory dizininde bir çözüm dosyası oluşturur.

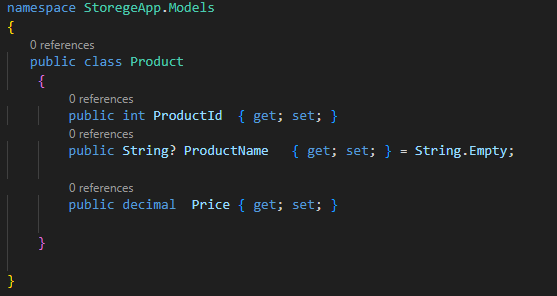
Solutiona oluşturduğumuz dotnet şablonlarını eklemek için dotnet sln –help komutundan faydalınabilir. dotnet sln <SLN\_FILE> add <PROJECT\_PATH>

Eklenen şablonlar/projeler dotnet sln .\Store.sln list komutu ile gürlüebilir.

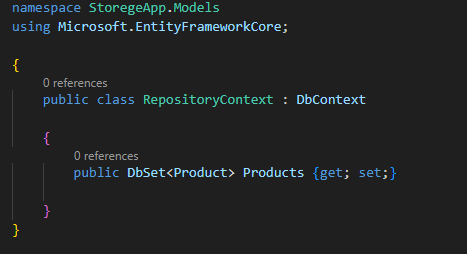


**Entity Framework (EF)**, Microsoft tarafından geliştirilen bir **ORM (Object-Relational Mapping)** aracıdır. ORM, veri tabanındaki tablo ve kayıtları, programlama dilindeki nesnelerle eşleştiren bir tekniktir. EF, .NET uygulamalarında veri işlemlerini kolaylaştırır ve geliştiricilerin SQL sorguları yazmak yerine, veri tabanı işlemlerini C# gibi dillerdeki nesnelerle yapmalarına olanak tanır.

Yukarıdaki görselden anlaşılacağı üzere EF bir veri tabanını soyutlama katmanı gibidir. İlk başta veri tabanı tablosunu temsil edecek bir sınıf (entity) oluşturulur ve ardından ona bağlı olarak DbContext oluşturulur . DbContext, EF Core'un veri tabanı ile etkileşim kurmasını sağlar. Veri tabanı bağlantı bilgilerini ve tabloları burada tanımlarsınız.



Şekil Product sınıfı



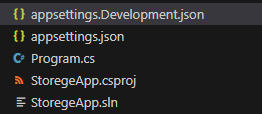
Şekil : DbContext

* **RepositoryContext Sınıfı**: Bu sınıf, EF Core'un DbContext sınıfından türetilmiştir ve veri tabanı ile etkileşime geçmek için kullanılır.
* **DbSet<Product> Products**: Bu özellik, Product sınıfına karşılık gelen veri tabanı tablosunu temsil eder(Products). Products koleksiyonu(tablosu), veri tabanında Product türündeki verileri, kayıtları içerir.(ProductId, ProductName ve Price)

DbSet; Entity Framework gibi Object-Relational Mapping (ORM) araçlarında, veri tabanındaki bir tabloya karşılık gelen bir koleksiyonu temsil eder.

**Connection String**, bir uygulamanın bir veri tabanına nasıl bağlanacağını tanımlayan, yapılandırılmış bir metin dizesidir. Bu dize, veri tabanına bağlantı kurulması için gereken çeşitli bilgileri içerir. Connection String, veri tabanının türüne (SQL Server, MySQL, Oracle, vb.), erişim yöntemine ve uygulamanın bağlanacağı veri tabanının bulunduğu ortama göre değişiklik gösterebilir. Yani yukarıdaki işlemleri yaptıktan sonra (Entity,DbSet) connection string bizim için veri tabanına bağlanmamızı sağlayacak bir anahtardır.

Bu aşamada C#’da appsetting.json ve appsetting.Development.json dosyalarından bahsetmek gerekirse



appsettings.json, C# ve .NET Core/ASP.NET Core projelerinde yaygın olarak kullanılan bir yapılandırma (configuration) dosyasıdır. Bu dosya, uygulamanızın çeşitli ayarlarını (örneğin, veri tabanı bağlantı bilgileri, API anahtarları, özel ayarlar vb.) saklamak için kullanılır. JSON formatında yazıldığı için, yapılandırma ayarlarını hiyerarşik olarak organize edebilir ve kolayca okuyabilir veya düzenleyebilirsiniz.

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

* C#'ta base anahtar kelimesi, türetilmiş bir sınıftan (derived class) temel sınıfa (base class) erişim sağlamak için kullanılır.
* Generic ifadeler(genel tipler), farklı türlerle çalışan sınıflar, yapılar, arabirimler ve yöntemler tanımlamanıza olanak tanıyan bir özelliktir. Yani herhangi bir türde çalışabilen metotlar oluşturmanızı sağlar. Tür, metodun çağrıldığı sırada belirtilir ”<>” arasında belirtilir. Bu sayede ayrı ayrı metodlar tanımlanaıza gerek kalmaz.

**DOTNET TOOLs**

dotnet tool, .NET CLI (Komut Satırı Arayüzü) tarafından kullanılan bir komuttur ve .NET geliştiricilerinin ek araçları (tools) kurmasına, yönetmesine ve kullanmasına olanak tanır. Bu araçlar, proje geliştirme sürecini kolaylaştıran ve çeşitli işlevler sunan komut satırı tabanlı uygulamalardır.

* **Küresel Araçlar (Global Tools):** Bu araçlar, makine genelinde herhangi bir klasörde kullanılabilir. Örneğin, dotnet tool install -g <tool-name> komutuyla bir aracı küresel olarak kurabilirsiniz.
* **Yerel Araçlar (Local Tools):** Bu araçlar, belirli bir proje veya çözüm kapsamında kullanılabilir. dotnet tool install --local <tool-name> komutuyla bir aracı proje bazında kurabilirsiniz.

Toollar arasından **dotnet ef** ‘e bakacak olursak;

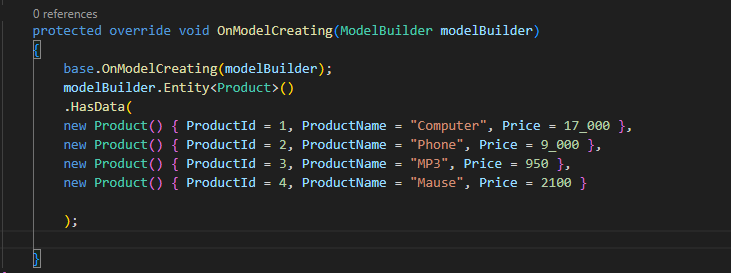
dotnet ef, Entity Framework Core (EF Core) için bir .NET CLI aracıdır. Bu araç, veri tabanı şemalarını yönetmek, veri tabanı işlemlerini otomatikleştirmek ve veri tabanı ile ilgili işlemleri gerçekleştirmek için kullanılır. EF Core, .NET Core uygulamalarında veri tabanı ile çalışmak için kullanılan bir ORM (Object-Relational Mapper) aracıdır.

Tüm bu aşamalardan sonra dotnet ef aracından migrations özelliğini kullanacağız.

Migrations, .NET frameworkünde kullanılan Entity Framework (EF) Core'da veri tabanı şemasını (yani tablolar, sütunlar, ilişkiler vb.) yönetmek ve bu şemadaki değişiklikleri veri tabanına uygulamak için kullanılan bir özelliktir. Migration'lar, veri tabanı ve uygulamanızdaki veri modelleri arasında senkronizasyon sağlar.

* dotnet ef migrations add <migration\_name> ile projede bir migrations oluşturulur. Bu migrations daha sonra (aşağıdaki komut ile) oluşturulacak veri tabanınının scriptlerini içerir
* dotnet ef database update komutu ile scriptler yansıtılır/uygulanır.

Çekirdek veri (seed data), yazılım geliştirme sürecinde önemli bir rol oynar, çünkü uygulamanın düzgün çalışabilmesi için belirli başlangıç verilerinin veri tabanında hazır bulunması gerekebilir. Bu verilerin otomatik olarak eklenmesi, özellikle veritabanı ilk oluşturulduğunda veya belirli bir migration (veri tabanı değişikliği) sırasında gerçekleşir. EF Core gibi ORM araçları bu işlemi kolaylaştırır ve HasData gibi yöntemlerle çekirdek verileri veri tabanına otomatik olarak ekler.



Örnek çekirdek veriler yukarıda gösterilmiştir.

**Dependency Injection (DI)**

Dependency Injection (DI), yazılım geliştirmede yaygın olarak kullanılan bir tasarım deseni ve prensibidir. Amaç, bir sınıfın ihtiyaç duyduğu bağımlılıkları (yani diğer sınıfları veya servisleri) dışarıdan sağlamaktır. DI, bir sınıfın diğer sınıflara olan bağımlılığını azaltarak, yazılımın esnekliğini, yeniden kullanılabilirliğini ve test edilebilirliğini artırır.

DI’nin Temel Amacı

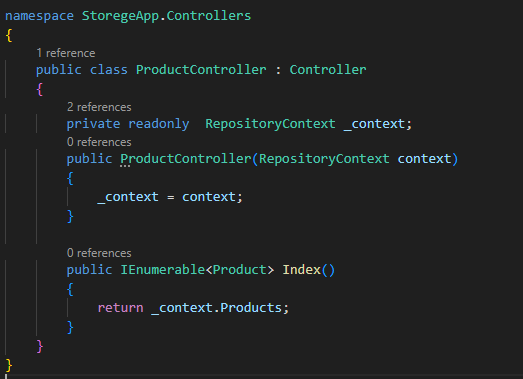
Bir yazılım sınıfı genellikle başka bir sınıfın işlevlerini kullanmak zorundadır. Bu durumda, ilk sınıf ikinci sınıfa bağımlı hale gelir. Bu tür bir bağımlılık, sınıfların birbiriyle sıkı bir şekilde bağlanmasına neden olur ve bu da kodun bakımını zorlaştırır. DI, bu bağımlılıkları kontrol altına alarak, sınıfların gevşek bağlı (loosely coupled) olmasını sağlar. Böylece, bağımlı nesnelerin oluşturulması ve yönetimi DI çerçevesi tarafından ele alınır, ve bu da kodun daha esnek ve sürdürülebilir olmasına katkıda bulunur.

DI olmadan veri tabanından bilgi döndüren kod:



Şekil : DI olmadan

Yukarıda görüldüğü gibi RepositoryContext DbContectOptionsBuilder’a doğrudan sıkı bağımlıdır.



Şekil :DI ile

* **private readonly RepositoryContext \_context**: Bu satır, RepositoryContext türünde özel bir alan tanımlar. readonly anahtar kelimesi, bu alanın sadece tanımlandığı yerde veya constructor'da (yapıcı metot) değer alabileceğini belirtir. \_context, veri tabanı ile iletişim kurmak için kullanılacak bir nesne olacaktır.
* **public ProductController(RepositoryContext context)**: Bu, ProductController sınıfının constructor'ıdır. Constructor, sınıfın bir örneği oluşturulduğunda ilk çalışan metottur.
* **\_context = context;**: Bu satırda, constructor'a dışarıdan (Dependency Injection ile) sağlanan context nesnesi, \_context adlı özel alana atanır. Böylece, bu sınıfın diğer metotları context nesnesini kullanabilir.
* **public IEnumerable<Product> Index()**:Bu metot, Product türündeki nesnelerin bir listesini döndürür (IEnumerable<Product>).
* **return \_context.Products;**: Bu satır, \_context nesnesi aracılığıyla veri tabanındaki Products tablosundaki tüm ürünleri alır ve döndürür.

**INVERSION of CONTROL ( IoC):**

Middlewarelerde kullanılan servislerin çözümlenebilmesi için IoC kaydının yapılması gerekir.

IoC, bir yazılım bileşeninin bağımlılıklarını (yani başka bileşenlere olan ihtiyaçlarını) kendisi oluşturmak yerine, dışarıdan verilmesini sağlayan bir tasarım desenidir. Bu, yazılımın daha modüler, test edilebilir ve bakımı kolay olmasını sağlar.

**Örnek:**

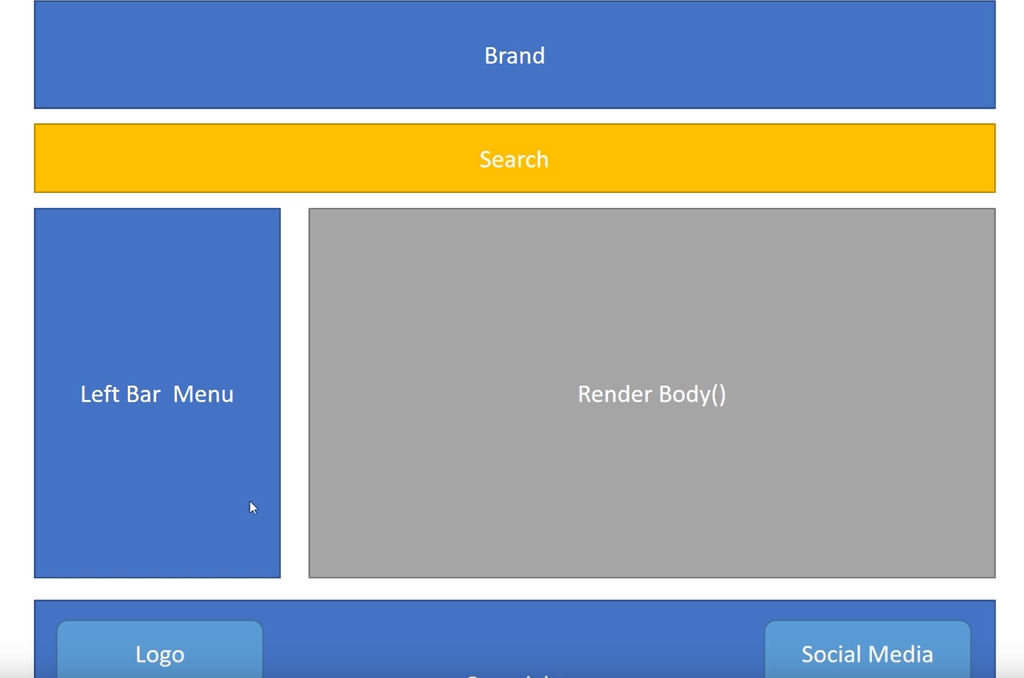
Diyelim ki bir e-ticaret uygulamanız var ve ürün bilgilerini bir veri tabanından çekiyorsunuz. Normalde, ürün bilgilerini çekmek için veri tabanı bağlantısını kuran bir sınıf yazabilirsiniz. Ancak IoC kullanarak, bu bağlantıyı dışarıdan sağlayabilirsiniz. Böylece, farklı veri tabanlarını kullanmak istediğinizde sadece bağlantı nesnesini değiştirmeniz yeterli olur.

DI, IoC’nin temelini oluşturur.

https://chatgpt.com/share/67af9421-cfc4-800e-9960-088c90d13f47

**LAYOUT**

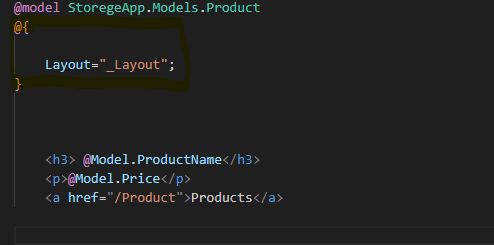
Layout, web uygulamalarında sayfanın veya uygulamanın genel yapısını, düzenini ve yerleşimini belirleyen bir tasarım yapısıdır. Kullanıcı arayüzünün (UI) nasıl organize edileceğini tanımlar ve içeriklerin (metin, görseller, videolar, formlar vb.) sayfa üzerinde nasıl yerleştirileceğini belirler. Layout, kullanıcıların site veya uygulama ile etkileşimlerini kolaylaştırmak, estetik bir görünüm sağlamak ve kullanıcı deneyimini optimize etmek amacıyla tasarlanır.



Render terimi, genellikle bir bileşenin veya içeriğin çıktı olarak gösterilmesini ifade eder.

ASP.NET MVC'de layout'lar, Razor görünüm motoru kullanılarak tanımlanır. Layout dosyası, sayfa tasarımının temel yapısını tanımlar ve sayfalar arasında ortak olan öğeleri içerir (örneğin, başlık, menü, altbilgi). Layout dosyaları genellikle Views/Shared klasöründe .cshtml uzantılı dosyalar olarak bulunur.

Her bir .cshtml dosyasında, layout dosyasını kullanmak için @{ Layout = "\_Layout.cshtml"; } ifadesini eklemeniz gerekir. Bu ifade, o sayfanın belirtilen layout dosyasını kullanmasını sağlar.Bir uygulama için birden çok layouttanımlanabilir.



@RenderBody() ifadesi, her sayfanın içeriğinin yerleştirileceği alanı belirtir. Kısacası, bu metot, layout dosyasının içinde sayfa içeriğinin nereye geleceğini belirler.

@RenderSection(): Layout dosyasında belirli bölümler (sections) tanımlamanızı sağlar. Bu bölümler, sayfalarda isteğe bağlı olarak kullanılabilir. Sayfalar arasında belirli içeriklerin, örneğin ek JavaScript dosyalarının veya ekstra stil dosyalarının yönetilmesi için kullanılır.

Yukarıdaki \_Layout,cshtml’i her cshtml dosyasına @{ Layout = "\_Layout.cshtml"; } şeklinde manuel olarak eklemek yerine tek bir yerden ekleyebiliriz. Bunun için Views/\_ViewStart.cshtm dosyasına tanımladığımız layout tanımları otomatik olarak tüm Razor pages cshtml dosyalarında geçerli olacaktır.

**Önemli!** İstenilen sayfada layout değerli null yapılarak o sayfa özelinde layout geçersiz kılınabilir.

**Önemli!** \_Navbar.cshtml ile uygulama navigasyonu sağlanabilir. Uygulama genelinde kullanım için \_Layout.cshtml’ e <partial name="\_Navbar" /> şeklinde bir ekleme yapılabilir. (Model olmadan kullanılan partial view’a örnektir)

***PARTIAL VİEW***

Partial View (Kısmi Görünüm), bir web sayfasının parçasını tanımlayan bir yapıdır. Genellikle, bir sayfanın sadece bir bölümünü oluşturmak için kullanılır. Bu, kod tekrarını önler ve büyük sayfaların yönetimini kolaylaştırır. Partial View'ler genellikle aşağıdaki durumlarda kullanılır:

1. **Kod Yeniden Kullanımı:** Aynı HTML ve kod parçalarını birden fazla sayfada kullanmak istiyorsanız, Partial View'ler kullanışlıdır. Örneğin, bir menü veya kullanıcı profil bilgisi birden fazla sayfada aynı şekilde görünebilir.
2. **Bölünmüş Görünüm Yönetimi:** Büyük ve karmaşık sayfaları daha yönetilebilir parçalara böler. Bu, hem geliştiricilerin kodu anlamasını kolaylaştırır hem de bakımını daha basit hale getirir.
3. **Dinamik İçerik Güncellemeleri:** Partial View'ler, sayfanın bir kısmını dinamik olarak güncellemeye ihtiyaç duyduğunuzda kullanışlıdır. Örneğin, kullanıcı bilgilerini AJAX kullanarak güncelleyebilir ve sadece ilgili kısımları yenileyebilirsiniz.

**Önemli!** Partial viewı ayrıca ilgili controllerın görünümüne ekleyerek sadece ilgili viewde görünümü düzenleyebiliriz. Bu durumda partial view modele bağlı olarak işlevsellik sağlayacaktır.

Birden fazla viewın kullanacağı bir görünüm varsa bu Shared altına aksi halde de ilgili view altına eklenerek partial view sağlanır.

**LİBMAN ve İSTEMCİ TARAFLI KÜTÜPHANELER:**

İstemci taraflı kütüphaneler, web tarayıcılarında çalıştırılan ve kullanıcı arayüzleri oluşturmayı, verilerle etkileşime geçmeyi ve kullanıcı deneyimini iyileştirmeyi sağlayan yazılım araçlarıdır.

**Libman** (Library Manager), özellikle ASP.NET projelerinde kullanılan bir kütüphane yönetim aracıdır. İstemci tarafı kütüphanelerini yönetmek ve projeye dahil etmek için tasarlanmıştır.

**Libman'in Sağladığı İmkanlar ve Özellikler**

1. **Kütüphane Yönetimi**:
   * Libman, JavaScript, CSS ve diğer istemci tarafı kütüphanelerini yönetmek için kullanılır. Bu kütüphaneler genellikle web projelerinde kullanılan harici kaynaklardır.
2. **CDN Desteği**:
   * Libman, Content Delivery Network (CDN) üzerinden kütüphaneleri projeye dahil etmenize olanak tanır. Bu, kütüphanelerin en güncel sürümlerini hızlı bir şekilde projeye entegre etmeyi sağlar.
3. **Yerel Dosya Desteği**:
   * Libman, kütüphaneleri yerel dosyalar olarak da yükleyebilir. Bu, kütüphaneleri doğrudan projeye ekleyerek, çevrimdışı veya özel bir yapılandırma gerektiren projeler için faydalıdır.
4. **Yalın Yapı**:
   * Libman, kütüphane yönetimi konusunda daha yalın ve hafif bir yaklaşım sunar. Bu, NuGet gibi diğer paket yöneticilerine kıyasla daha az karmaşıklık ve daha basit bir yapı sağlar.
5. **Konfigürasyon Dosyası**:
   * Kütüphanelerin nasıl ve nereden alınacağını belirleyen bir konfigürasyon dosyası (libman.json) ile çalışır. Bu dosya, kütüphanelerin kaynağını ve hangi dosyaların alınacağını belirler.

Özellikle çeşitli bilindik Jquery, Bootstrap, Font-Awesome, Angular ve daha fazla istemci tarafli kütüphaneler projelerimize dahil edilerek çok daha kullanıcı dostu arayüzler çok daha hızlı bir şekilde oluşturulur. Libman ise bunu ASP.NET Core için yapan tooldur.

Yerel Kütüphaneler Projelere Nasıl Dahil Edilir?

**1. CDN (Content Delivery Network) Kullanımı**

CDN'ler, istemci taraflı kütüphaneleri doğrudan HTML dosyalarınıza eklemenizi sağlar. Bu yöntem, kütüphanelerinizi hızlı bir şekilde entegre etmenin kolay bir yoludur.

**Örnek:**

\_Layout.cshtml veya ilgili HTML dosyanıza kütüphaneyi ekleyebilirsiniz:

* <!-- JavaScript kütüphanesi -->

<script src="https://cdn.jsdelivr.net/npm/jquery@3.6.0/dist/jquery.min.js"></script>

* <!-- CSS kütüphanesi -->

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">

**2.LİBMAN**

LibMan ile kütüphane ekleyin. wwwroot dizinine kütüphaneleri ekledikten sonra yine yukarıdaki gibi ilgili HTML’lere ekleyebilirsiniz.

* <!-- JavaScript kütüphanesi -->

<script src="~/lib/library-name/library.js"></script>

* <!-- CSS kütüphanesi -->

<link rel="stylesheet" href="~/css/library-name.min.css">

CDNJS: <https://cdnjs.com/> üzerinden kütüphaneler seçilip projemize dahil edilebilir.

CDNJS, **Cloudflare** altyapısını kullanır ve bu sayede dünya çapında hızlı, güvenli ve kesintisiz bir içerik dağıtım hizmeti sağlar. CDNJS, **jQuery, Bootstrap, React, Angular, Vue.js, FontAwesome** gibi yaygın kullanılan JavaScript ve CSS kütüphanelerini barındırır. Kütüphanelerin çeşitli sürümleri mevcuttur, böylece en güncel sürümleri kullanabileceğiniz gibi daha eski sürümlere de erişebilirsiniz.