

C#'ta Attribute Nedir? Nasıl Kullanılır ve Kendi Attribute'unuzu Nasıl Oluşturursunuz?

6 min read · Apr 30, 2025



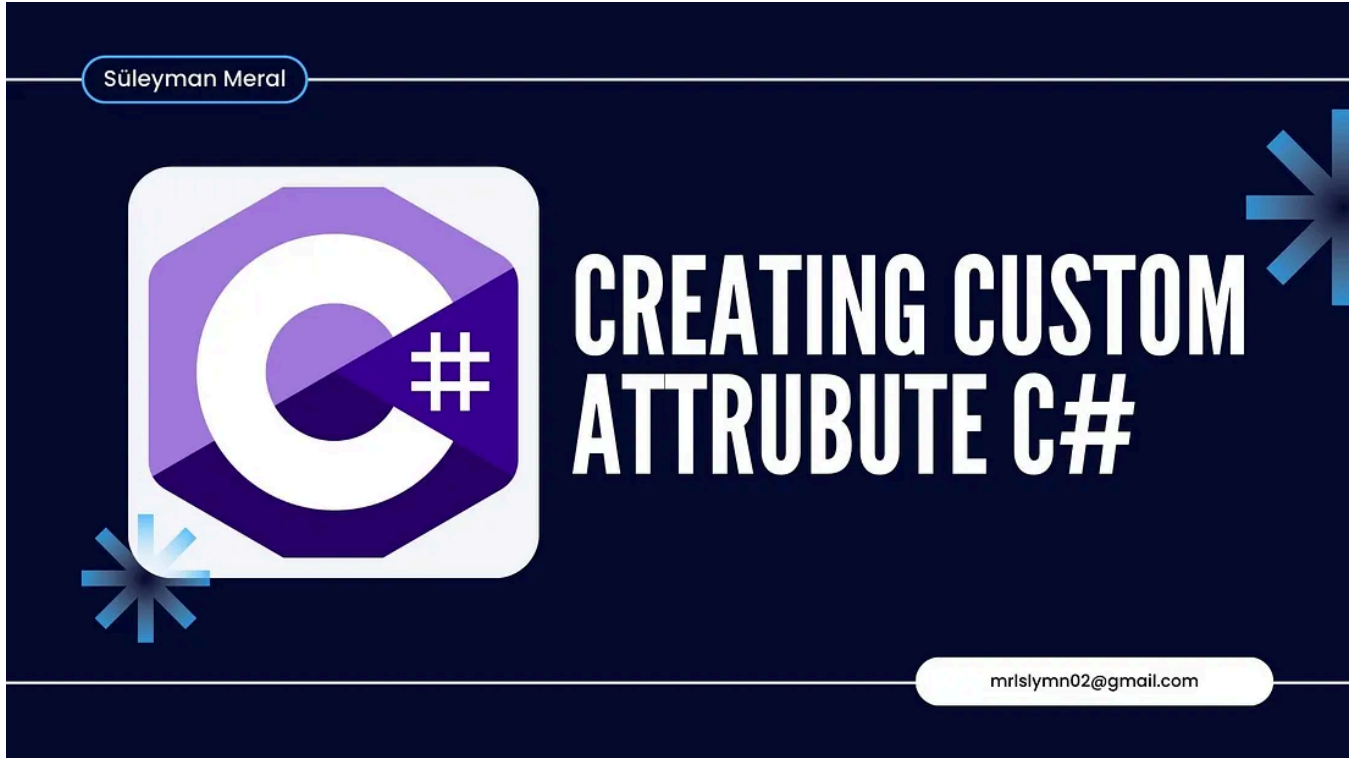
Süleyman Meral



Share



More



Herkese Merhaba! Bu yazımda sizlere hepimizin kod satırlarında gördüğü fakat ne işe yaradığı hakkında pek bilgi sahibi olunmayan attribute kavramından bahsedeceğim. Daha sonra bir örnek ile kendi attribute yapımızı oluşturacağız. Ve bu yapının temiz ve solide uygun kod yazmamızı nasıl sağladığını göreceğiz. Şimdiden iyi okumalar.

C# programlama dilinde `Attribute` kavramı, kodlara **meta-veri (metadata)** eklememizi sağlayan güçlü bir araçtır. Derleyiciye, çalışma zamanına veya diğer geliştiricilere belirli davranışları tanımlamak için kullanılır. Attribute'lar sayesinde

sınıflara, metodlara, özelliklere hatta parametrelere açıklayıcı bilgiler ekleyebiliriz. Bu açıklamalar, derleme sırasında, hata kontrolünde veya uygulama çalışırken Reflection yardımıyla okunabilir. Örneğin [Obsolete] veya [Serializable] gibi yerleşik attribute'lar, C# geliştiricilerinin aşına olduğu örneklerdendir.

Katmanlı mimaride bir proje geliştirirken Entity property'lerinde de bu yapılara rastlayabiliriz. Aşağıda buna bir örnek verilmiştir.

```
2 references
public abstract record BookDtoForManipulation
{
    [Required(ErrorMessage = "Title is a required field")]
    [MinLength(2, ErrorMessage = "Title must consist of at least 2 characters")]
    [MaxLength(50, ErrorMessage = "Title must consist of at maximum 2 characters")]
    0 references
    public string? Title { get; init; }

    [Required(ErrorMessage = "Price is a required field")]
    [Range(10, 1000)]
    0 references
    public decimal Price { get; init; }
}
```

Görüldüğü üzere property'lerin özelliklerini tanımlamak için çeşitli attribute yapıları kullanılmış. Bu yapılar .NET'in bize sunduğu hazır yapılardır. Program runtime'da bu attribute yapılarına dikkat ederek işlem yapar. Yani reflection kullanır. Reflection hakkında bir bilginiz yoksa daha önce reflection hakkında yazdığım yazıyı inceleyebilirsiniz.(Bu kod parçası bir DTO ya ait olduğu için record type kullanılmıştır.)

.NET CORE Reflection(Runtime'da Kodu Okumak ve Değiştirmek: Reflection'ın Gücü)

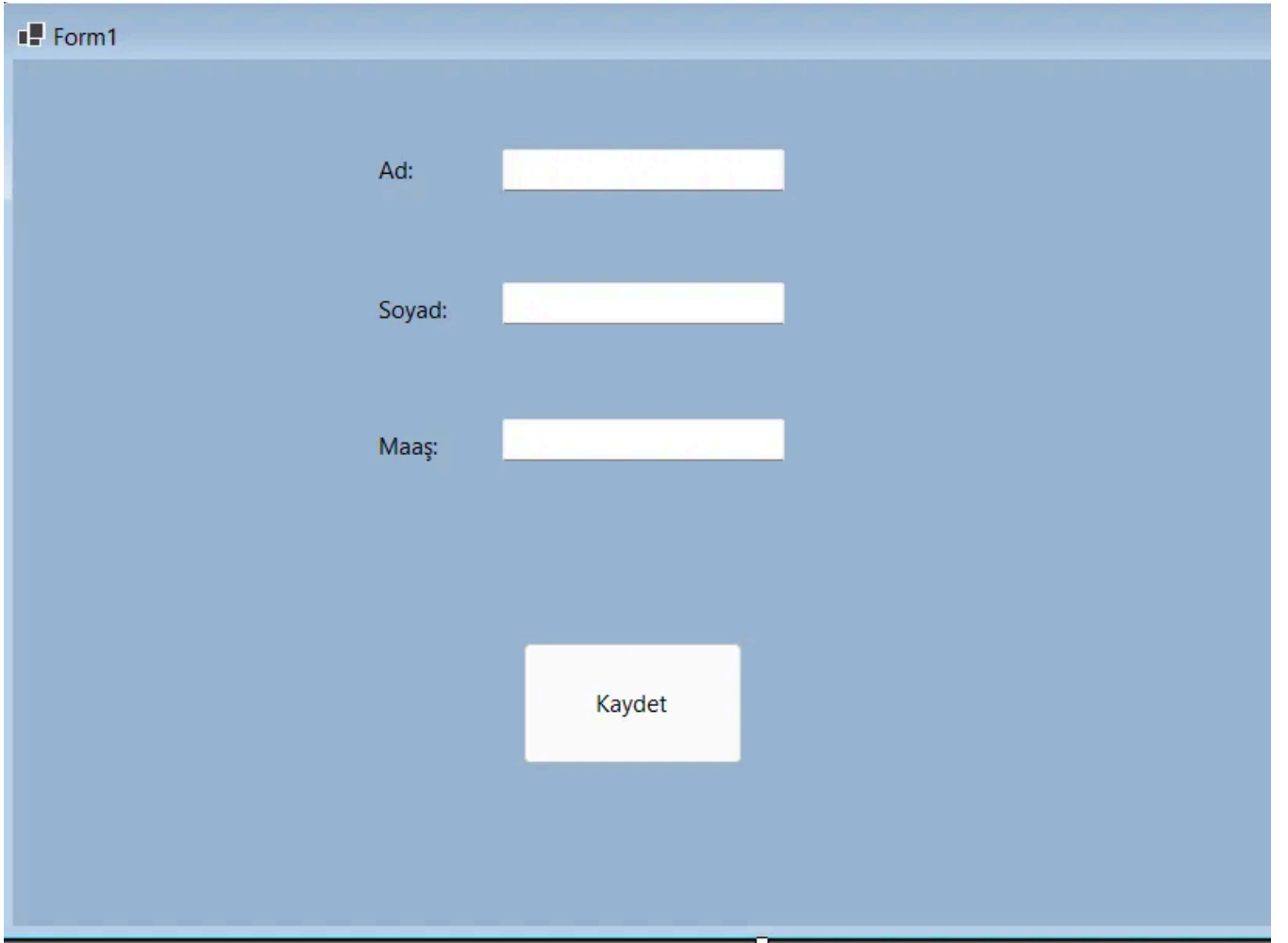
Herkese merhaba! Bu yazımda sizlere Reflection kavramından bahsedeceğim. Reflection Nedir? Niçin Kullanılır...

medium.com

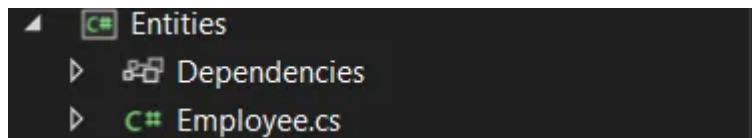
Yukarıda hazır olarak gelen attribute yapılarını gördük. Peki bizim bir attribute oluşturup onu kodumuzda kullanmamız mümkün mü? Evet bu işlemi kendimiz de yapabiliriz. Şimdi hep birlikte bir proje oluşturalım ve bu proje içerisinde kendi attribute yapımızı yazalım. Daha sonra bunu projemizde kullanalım.

Örnek bir senaryo üzerinden gidelim. Çalıştığımız yazılım şirketine bir iş gelmiş olsun. Bu işin tanımında bir fabrikanın yeni gelen işçilerin ad soyad ve maaş bilgilerinin kayıt edilmesi için bir program yazmamız istensin.

Örnek olarak bunu bir form uygulamasında yapalım.

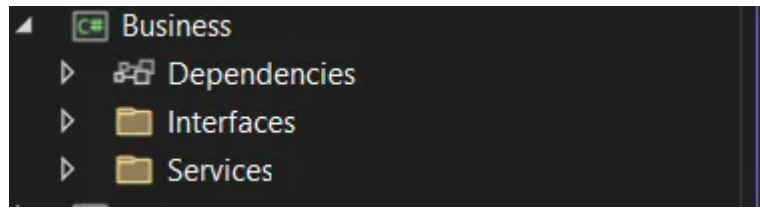


Arayüzümüzü bu şekilde tasarlamış olalım. Şimdi ise katmanlı mimariye uygun bir proje oluşturmak için Entities katmanı oluşturalım ve içerisine Employee adında bir class ekleyelim.



```
public string Name { get; set; }  
  
public string SurName { get; set; }  
  
public int Salary { get; set; }
```

Employee classımız bu 3 özelliği barındıracak. Çünkü bizden bu 3 özelliği kaydetmemiz istendi.



Daha sonra bir business katmanı oluşturalım. İçerisine Interfaces ve Services klasörlerini ekleyelim.

```
public interface IEmployeeService
{
    string SaveEmployee(Employee emp);
}
```

```
public class EmployeeService : IEmployeeService
{
    public string SaveEmployee(Employee emp)
    {
        return $"Ad: {emp.Name} Soyad: {emp.SurName} Maaş: {emp.Salary} kaydedildi";
    }
}
```

Uygulamamız göstermelik olduğu için veritabanı eklemeyeceğiz.

Form1 de butonumuzun click fonksiyonunu yazalım.

```
if (string.IsNullOrEmpty(txtName.Text))
{
    MessageBox.Show("Ad kısmı boş bırakılamaz.");
}
else if (string.IsNullOrEmpty(txtSurname.Text))
{
    MessageBox.Show("Soyad kısmı boş bırakılamaz.");
}
else if (string.IsNullOrEmpty(txtSalary.Text))
{
    MessageBox.Show("Maaş kısmı boş bırakılamaz.");
}
```

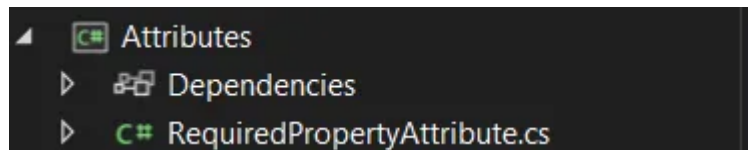
```
        MessageBox.Show("Maaş kısmı boş bırakılamaz.");
    }
    emp.Name = txtName.Text;
    emp.SurName = txtSurname.Text;
    emp.Salary = int.Parse(txtSalary.Text);

    var response = es.SaveEmployee(emp);
    MessageBox.Show(response);
```

Kodda görüldüğü gibi ilgili inputun boş olup olmadığını sürekli kontrol etmemiz gerekiyor. Birden fazla input boş olduğunda daha fazla kontrol yazmamız gerekecek. Uygulamamız şimdilik bu şekilde teslim etmiş olalım. İlgili şirket bizden çalışan kaydı yapacağı zaman telefon numarasını da kayıt edeceğini söylesin. Ve bizden projeyi güncellemeyi istesin.

Yapmamız gereken ilk iş employee classına yeni bir property eklemek olacak. Fakat işler buradan sonra karışıyor. Her yeni property eklediğimizde kontrol sayısı iyice artacak. Ve en önemli kısım olarak SOLID 'in Open/Closed prensibini ezmiş olacağız. Yani kodumuz değişime açık olmuş oldu. Bizim kodumuzu değişime kapalı gelişime açık hale getirmemiz gerekiyor. Bu noktada da devreye attribute yapılarımız girecek.

Bizim sorunumuz property ile ilgili. Propertyler için bir attribute oluşturursak bu sorunun önüne geçmiş olacağız. Peki bunu nasıl yapacağız? Hep beraber inceleyelim.



Bir attribute katmanı oluşturuyorum ve içine RequiredPropertyAttribute adında bir class ekliyorum. Bir attribute classı oluştururken sonunu Attribute ile bitirmeliyiz.

```
public class RequiredPropertyAttribute:Attribute
{
    public RequiredPropertyAttribute(string errorMessage)
    {
        ErrorMessage = errorMessage;
    }
}
```

```
    public string ErrorMessage { get; set; }  
}
```

Classımız Attribute classından türemek zorunda. Ayrıca attribute yapısının içine bir mesaj ekliyoruz. Bu mesaj kullanıcıya gösterilen mesaj olacak. Farklı kontroller yapmak istersek bunu çeşitlendirebiliriz.

Attribute yapımız tamam. Şimdi bunun kontrolünü yapacak kodumuzu yazalım. Burada devreye Reflection girecek.

```
public class ValidationControl  
{  
    public static string Validate(object obj)  
    {  
        StringBuilder errorMessages = new StringBuilder();  
  
        Type type = obj.GetType();  
  
        foreach( var property in type.GetProperties())  
        {  
            var attributes = (RequiredPropertyAttribute)Attribute.GetCustomAttr  
  
            if(attributes is not null)  
            {  
                var value = property.GetValue(obj);  
                if (value == null || (value is string str && string.IsNullOrEmpty(str)) )  
                {  
                    string errorMessage = attributes.ErrorMessage ?? $"{property.Name} is required";  
                    errorMessages.AppendLine(errorMessage);  
                }  
            }  
        }  
        return errorMessages.ToString();  
    }  
}
```

Doğrulama kontrolü yaptığımız class bu şekilde. İçinde statik yapılı (nesne oluşturmadan class ile erişilebilen) bir method mevcut. Şimdi kodları teker teker açıklayalım.

```
StringBuilder errorMessages = new StringBuilder();
```

StringBuilder yapılı bir errorMessages değişkeni oluşturduk. Hata mesajları çok olabileceği için bu yapıyı tercih ettik.


```
Type type = obj.GetType();
```

type değişkenine metodumuza girilen objenin türünü atadık. (Örneğin Employee türünde bir obje gelirse type Employee olacak.

```
foreach( var property in type.GetProperties())
```

Döngü işe type değişkeninin içerisindeki Property'lere ulaşıyoruz. Bunları bize reflection sağlıyor.

```
var attributes = (RequiredPropertyAttribute)Attribute.GetCustomAttribute(proper
```



RequiredProperty attribute yapısını içeren property'leri attributes değişkenine atadık.

```
if(attributes is not null)
{
    var value = property.GetValue(obj);
    if (value == null || (value is string str && string.IsNullOrEmpty(str)))
    {
        string errorMessage = attributes.ErrorMessage ?? $"{property.Name} is required";
        errorMessages.AppendLine(errorMessage);
    }
}
```

```
}  
}
```

Attribute mevcut ise o attribute yapısını içeren propertynin değerini alıyoruz. (Örneğin kullanıcı arayüzde Ad kısmına “Beyza” yazdı. Bu kısma karşılık gelen prop Name olduğu için Name property’sinin değerini alıyor.

```
if (value == null || (value is string str && string.IsNullOrEmpty(str)))  
{  
    string errorMessage = attributes.ErrorMessage ?? $"{property.Name} alanı boş bırakılamaz."  
    errorMessages.AppendLine(errorMessage);  
}
```

Asıl işlemimiz burada gerçekleşiyor. Eğer value null ise veya boşluk içeriyorsa attribute içinde tanımlı ErrorMessage değerine ulaşıyoruz. Eğer ErrorMessage tanımlı değilse default olarak \$"{property.Name} alanı boş bırakılamaz.” yazılıyor. Bu döngü bu attribute yapısını içeren tüm propertyler için tekrarlanıyor ve hata mesajı son olarak errorMessages değişkenine atanıyor. Ve bu değeri return ediyoruz. Biz burada null kontrolü yaptık. Fakat bu istediğimiz gibi şekillenebilir. Örneğin çalışanların TC numarasını da girmeleri istenirse belli bir kurala göre attribute oluşturulabilir.

```
[RequiredProperty("İsim Boş Olamaz")]  
public string Name { get; set; }  
  
[RequiredProperty("Soyad Boş Olamaz")]  
public string SurName { get; set; }  
  
[RequiredProperty("Maaş Boş Olamaz")]  
  
public int Salary { get; set; }  
[RequiredProperty("Telefon Boş Olamaz")]  
  
public string Phone { get; set; }
```


Attribute yapılarımızı propertylerden önce bu şekilde tanımlıyoruz.

```
emp.Name = txtName.Text;
emp.SurName = txtSurname.Text;
emp.Salary = int.Parse(txtSalary.Text);
emp.Phone = txtPhone.Text;

var errorMessage = ValidationControl.Validate(emp);

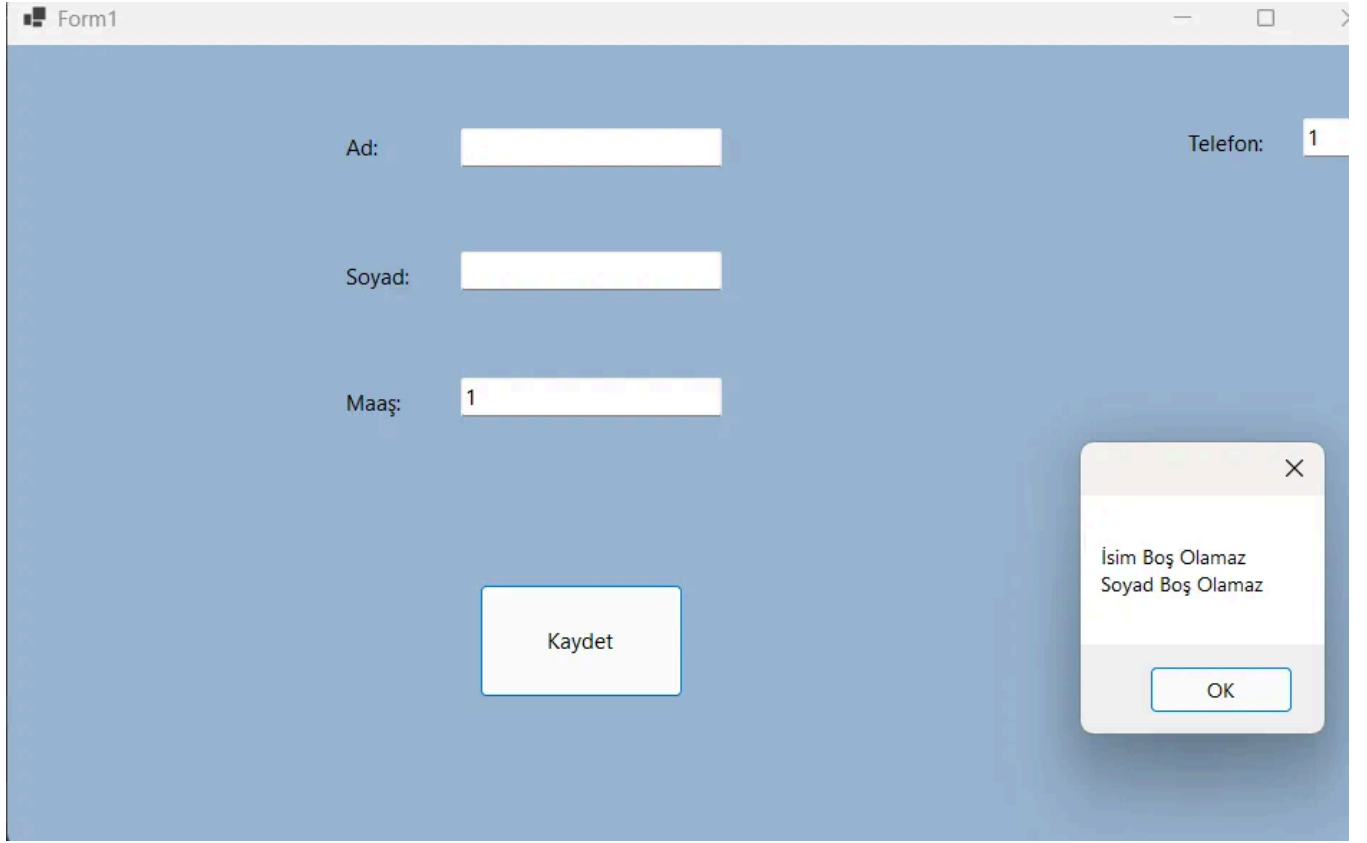
if (!string.IsNullOrEmpty(errorMessage))
{
    MessageBox.Show(errorMessage);
    return;
}

var response = es.SaveEmployee(emp);
MessageBox.Show(response);
```

Kaydet butonumuzun kodlarını bu şekilde güncelliyoruz. Diğer kontrole kıyasla hem SOLID'i ezmemiş olduk hem daha az kod yazmış olduk. Oluşturulan nesne ValidationControl classındaki Validate fonksiyonuna veriliyor. Bu fonksiyon ilgili Attribute yapılarını ulaşıp hataları kontrol ediyor.

The screenshot shows a Windows application window titled "Form1" with a blue background. The form contains three input fields: "Ad:" (Name), "Soyad:" (Surname), and "Maaş:" (Salary). The "Ad:" field is empty, "Soyad:" contains "Meral", and "Maaş:" contains "60000". There is a "Telefon:" (Phone) label and a text box containing "123". A "Kaydet" (Save) button is at the bottom center. An error message dialog box is open in the bottom right corner, displaying the text "İsim Boş Olamaz" (Name cannot be empty) and an "OK" button.

Burada çıkan hata mesajı [RequiredProperty(“İsim Boş Olamaz”)] burada belirttiğimiz hata mesajıdır.



İstedığımız kadar prop ekleyelim buttonumuzun fonksiyonunu değiştirmeden validation işlemini gerçekleştirebiliriz. Kodu daha temiz hale ve solide uygun hale getirmek için çeşitli işlemler yapılabilir. Ben sadece attribute yapısının önemini vurgulamak amacı ile bu yolu izledim. Single Responsibility prensibini ezmek adına validation işlemi de başka yerlere taşınabilir.

Bu yazıda, .NET ortamında kendi attribute yapımızı oluşturarak nasıl daha esnek ve sürdürülebilir bir doğrulama sistemi kurabileceğimizi örnek bir senaryo üzerinden inceledik. SOLID prensiplerine uygun, geliştirilebilir bir mimari oluşturmanın temel adımlarını attık. Konuyu daha iyi pekiştirmek isterseniz aşağıdaki videodan detaylı anlatımı izleyebilirsiniz. 🙌

<https://www.youtube.com/watch?v=5WEnoI-JL9U&t=1430s>

Süleyman Meral

.NET Developer/Software Engineering Student

Yazılım

Net Core

C Sharp Programming



Edit profile

Written by Süleyman Meral

6 followers · 6 following

Backend Developer(.NET Core)

No responses yet



Süleyman Meral

What are your thoughts?

More from Süleyman Meral