

Unit Test nedir ?

Unit Test, bir yazılımın en küçük test edilebilir bölümlerinin, tek tek ve bağımsız olarak doğru çalışması için incelendiği bir yazılım geliştirme sürecidir. Unit Test yazılım testinin ilk seviyesidir ve entegrasyon testinden önce gelir. Unit Testleri geliştiriciler kendileri yazar ve yürütürler.

Neden Unit Test yaparız ?

Buradaki amacımız yazılımın her biriminin tasarlandığı şekilde gerçekleştiğini doğrulamaktır. Unit Test yazmak kodda yeniden düzenleme(Refactor) işlemini yapmayı kolaylaştırır. Kodda değişiklik yaptığımızda, Unit Testi çalıştırıp oluşturduğumuz algoritmaya uygun bir şekilde çalışıp çalışmadığını kolaylıkla test edebiliriz. Unit Test' ler tüm hataları ortaya çıkarmaz, çünkü her parça izole şekilde test edilmekte ve entegrasyon yapıldığında her şeyin düzenli çalışacağı anlamına gelmez.

Bazı Unit Test Framework' leri

1. Robot Framework
2. JUnit
3. Spock
4. NUnit
5. TestNG
6. Jasmin
7. Mocha

Unit Test nasıl yazılır?

Geliştireceğimiz yazılımları yazmadan önce Unit Test' lerini yazmamız gerekiyor ve Unit Test yazmak için de bazı kurallara uymamız gerekiyor.

1. En küçük parçacığı test edilmeli
2. Sadece bir senaryo test edilir.
3. Kullanılan adımlar belirlenir.
4. Test method ismi test edilen senaryonun yansıması olmalıdır.
5. Test edilen kısım diğer kısımlardan bağımsız olmalıdır.

6. Testlerimiz tam otomatik şekilde çalışmalıdır.
7. Hızlı çalışabilmeli ve çabuk sonuçlar vermelidir.
8. Okunaklı, anlaşılabilir ve sürdürülebilir olmalıdır.
9. Test başarısız olduğunda durmalı ve iyi bir hata raporu döndürmelidir. Bu hata raporunda neyi test ettin ? ne yapmalı ? beklenen çıktı neydi ve gerçekte ne yaptıdır ?

* Unit test yazmak Arrange, Act ve Assert olmak üzere üç aşamadan oluşur!

■ **Arrange**

Test edilecek metodun kullanacağı kaynakların hazırlandığı bölümdür. Değişken tanımlama, nesne oluşturma vs. gerçekleştirilir.

■ **Act**

Arrange aşamasında hazırlanan değişkenler yahut nesneler eşliğinde test edilecek olan metodun çalıştırıldığı bölümdür. Mümkün mertebe kısa ve öz olması makbuldür.

■ **Assert**

Act aşamasında yapılan testin doğrulama evresidir. Tek bir Act'te birden fazla sonuç gerçekleştirilebilir. Misal olarak; exception fırlatılabilir yahut herhangi bir türde result dönebilir.

Calculator Project Unit Testing

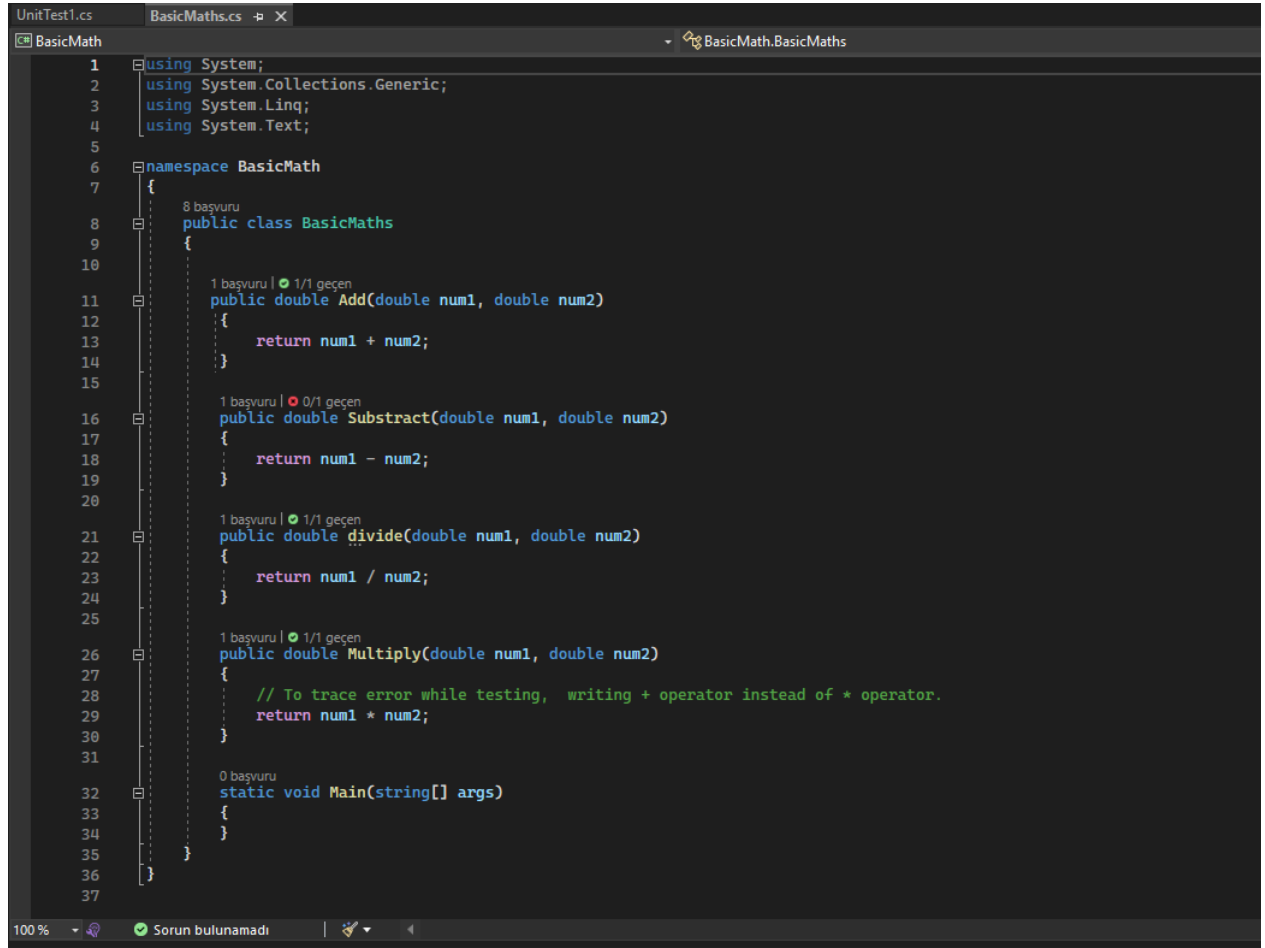
Aşağıda verilen kod örneği Matematik hesaplamalarının yapıldığı fonksiyonları içermektedir. Bu fonksiyonlar : Add(), Subtract(), Divide(), Multiply() 'dır.

Add() : Bu fonksiyon iki sayıyı toplamaya yarar.

Subtract() : Bu fonksiyon iki sayıyı birbirinden çıkartmaya yarar.

Divide() : Bu fonksiyon iki sayıyı birbirine bölmeyi sağlar.

Multiply() : Bu fonksiyon iki sayıyı birbirleri ile çarpar.



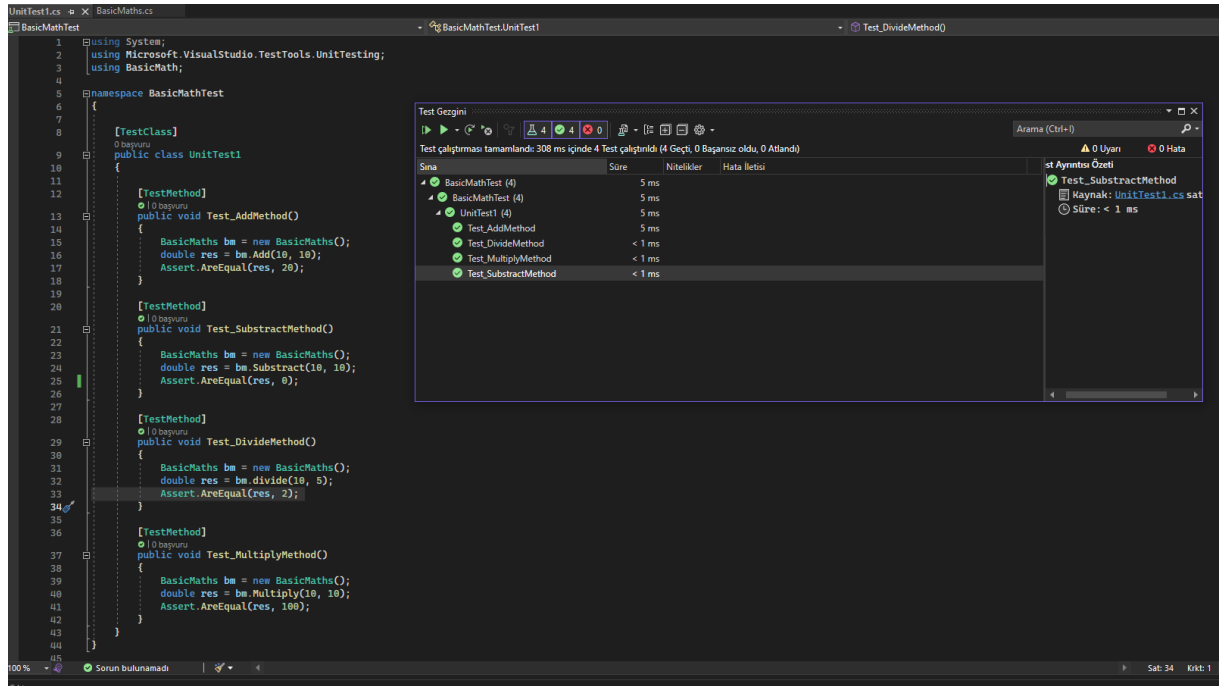
```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace BasicMath
7 {
8     8 başvuru
9     public class BasicMaths
10     {
11         1 başvuru | 1/1 geçen
12         public double Add(double num1, double num2)
13         {
14             return num1 + num2;
15         }
16         1 başvuru | 0/1 geçen
17         public double Subtract(double num1, double num2)
18         {
19             return num1 - num2;
20         }
21         1 başvuru | 1/1 geçen
22         public double Divide(double num1, double num2)
23         {
24             return num1 / num2;
25         }
26         1 başvuru | 1/1 geçen
27         public double Multiply(double num1, double num2)
28         {
29             // To trace error while testing, writing + operator instead of * operator.
30             return num1 * num2;
31         }
32         0 başvuru
33         static void Main(string[] args)
34         {
35         }
36     }
37 }
```

Aşağıda verilen BasicMath classı projemizin Unit Test kodlarını içermektedir.

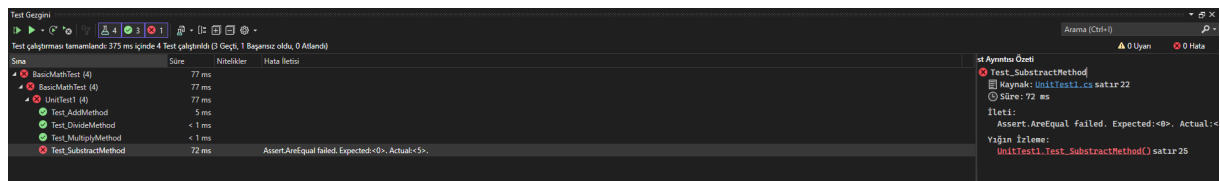
```
UnitTest1.cs x BasicMaths.cs
BasicMathTest
1 using System;
2 using Microsoft.VisualStudio.TestTools.UnitTesting;
3 using BasicMath;
4
5 namespace BasicMathTest
6 {
7
8     [TestClass]
9     public class UnitTest1
10    {
11
12        [TestMethod]
13        public void Test_AddMethod()
14        {
15            BasicMaths bm = new BasicMaths();
16            double res = bm.Add(10, 10);
17            Assert.AreEqual(res, 20);
18        }
19
20        [TestMethod]
21        public void Test_SubtractMethod()
22        {
23            BasicMaths bm = new BasicMaths();
24            double res = bm.Subtract(10, 10);
25            Assert.AreEqual(res, 5);
26        }
27
28        [TestMethod]
29        public void Test_DivideMethod()
30        {
31            BasicMaths bm = new BasicMaths();
32            double res = bm.divide(10, 5);
33            Assert.AreEqual(res, 2);
34        }
35
36        [TestMethod]
37        public void Test_MultiplyMethod()
38        {
39            BasicMaths bm = new BasicMaths();
40            double res = bm.Multiply(10, 10);
41            Assert.AreEqual(res, 100);
42        }
43    }
44
45 }
```

Bir önceki sayfada tanımladığımız fonksiyonların her biri için Test method fonksiyonlarını oluşturuyoruz. Fonksiyonları oluştururken kullanacağımız fonksiyon çağırılıp girilen değer doğrultusunda çıktı kontrol edilir. Bu kontrol Assert komutu tarafından yapılır.

Unit test kodlarımızı çalıştırdığımızda herhangi bir hata almıyorsa aşağıdaki gibi sonuç alınır. Burada yeşil tik olarak gördüklerimiz ,bir önceki sayfada yazmış olduğumuz test fonksiyonlarının testten başarıyla geçtiği anlamına gelmektedir.



Aşağıdaki görselde ise test fonksiyonlarında herhangi bir hata ile karşılaşırsak başarısız testler gözüktür. Bu hataların hangi fonksiyonda ve satır numarasında olduğu belirtilir.



Bank Account Project Unit Testing

BankAccount sınıfını ve banka hesapları için temel işlemleri (kredi, borç) gerçekleştirmek için iki yöntem içerir. BankAccount sınıfı, bir müşterinin adını ve hesap bakiyesini takip eder.

BankAccount sınıfında iki adet sabit değişken de tanımlanmıştır.

DebitAmountExceedsBalanceMessage ve

DebitAmountLessThanZeroMessage, kullanıcıya uygun hata mesajlarını göstermek için kullanılır.

BankAccount sınıfında, **Credit()** yöntemi, hesaba belirli bir miktar kredi eklemek için kullanılır ve **Debit()** yöntemi, hesaptan belirli bir miktar borç almak için kullanılır. Bu yöntemler, hesap bakiyesindeki değişiklikleri hesaplar.

Debit() yöntemi, işlem sırasında hesap bakiyesinin negatif olamayacağını veya girdiği miktarın hesap bakiyesinden daha büyük olamayacağını doğrulamak için birkaç koşul içerir. Eğer girdiği miktar hesap bakiyesinden daha büyükse veya girdiği miktar negatif ise bir hata mesajı gösterir.

Main() yöntemi, BankAccount sınıfının işlevselliğini test etmek için örnek bir kullanım gösterir. İlk olarak, bir müşteri adı ve başlangıç bakiyesi ile bir BankAccount nesnesi oluşturulur. Sonra Credit() ve Debit() yöntemleri, hesap bakiyesinde değişiklikler yapmak için kullanılır. Son olarak, hesap bakiyesi Current balance ise \$x.yy şeklinde ekrana yazdırılır.

```
Program.cs Nesne Tarayıcısı
Bank1Account.cs BankAccountNS.BankAccount

1 using System;
2
3 namespace BankAccountNS
4 {
5
6     /// <summary>
7     /// Bank account demo class.
8     /// </summary>
9     4 başvuru
10    public class BankAccount
11    {
12
13        private readonly string m_customerName;
14        private double m_balance;
15        public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds balance";
16        public const string DebitAmountLessThanZeroMessage = "Debit amount is less than zero";
17
18
19        0 başvuru
20        private BankAccount() { }
21
22        1 başvuru
23        public BankAccount(string customerName, double balance)
24        {
25            m_customerName = customerName;
26            m_balance = balance;
27        }
28
29        0 başvuru
30        public string CustomerName
31        {
32            get { return m_customerName; }
33        }
34
35        1 başvuru
36        public double Balance
```

```
Program.cs Nesne Tarayıcısı
Bank1Account.cs BankAccountNS.BankAccount

31 }
32
33 1 başvuru
34 public double Balance
35 {
36     get { return m_balance; }
37 }
38
39 1 başvuru
40 public void Debit(double amount)
41 {
42     if (amount > m_balance)
43     {
44         throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountExceedsBalanceMessage);
45     }
46
47     if (amount < 0)
48     {
49         throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountLessThanZeroMessage);
50     }
51
52     m_balance -= amount; // intentionally incorrect code
53 }
54
55 1 başvuru
56 public void Credit(double amount)
57 {
58     if (amount < 0)
59     {
60         throw new ArgumentOutOfRangeException("amount");
61     }
62
63     m_balance += amount;
64 }
65
66 0 başvuru
67 public static void Main()
68 {
69     Sorun bulunamadı
```

```
61 }
62
63 0 başvuru
64 public static void Main()
65 {
66     BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);
67
68     ba.Credit(5.77);
69     ba.Debit(11.22);
70     Console.WriteLine("Current balance is ${0}", ba.Balance);
71 }
72 }
```

Burada Bank Account dosyamızın Unit Testleri yer almaktadır.

İlk test metodu (**Debit_WithValidAmount_UpdatesBalance**) doğru bir şekilde para çekme işleminin yapılıp yapılmadığını test eder. Bu test, bir hesap oluşturur, belirli bir miktar para çekme işlemi yapar ve ardından hesap bakiyesinin doğru şekilde güncellenip güncellenmediğini doğrular.

İkinci test metodu

(**Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException**), Debit yönteminin, negatif bir miktar para çekme işlemi yaparken beklenen istisnayı fırlatıp fırlatmadığını test eder.

Üçüncü test metodu

(**Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException**),

Debit yönteminin, hesap bakiyesinin altında bir miktarda para çekme işlemi yaparken beklenen istisnayı fırlattığını test eder. Bu test, bir istisna fırlatıldığında, fırlatılan istisnanın beklenen istisnanın olup olmadığını kontrol eder.

```
TestProject3
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using BankAccountNS;
3  namespace BankTests
4  {
5      [TestClass]
6      public class BankAccountTests
7      {
8          [TestMethod]
9          public void TestMethod1()
10         {
11         }
12         [TestMethod]
13         public void Debit_WithValidAmount_UpdatesBalance()
14         {
15             // Arrange
16             double beginningBalance = 11.99;
17             double debitAmount = 4.55;
18             double expected = 7.44;
19             BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);
20
21             // Act
22             account.Debit(debitAmount);
23
24             // Assert
25             double actual = account.Balance;
26             Assert.AreEqual(expected, actual, 10, "Account not debited correctly");
27         }
28         [TestMethod]
29         public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRangeException()
30         {
31             // Arrange
32             double beginningBalance = 11.99;
33             double debitAmount = -100.00;
34         }
35     }
36 }
```



```
UnitTest1.cs Nesne Tarayıcısı
TestProject3 BankTests.BankAccountTests

31 {
32     // Arrange
33     double beginningBalance = 11.99;
34     double debitAmount = -100.00;
35     BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);
36
37     // Act and assert
38     Assert.ThrowsException<System.ArgumentOutOfRangeException>(() => account.Debit(debitAmount));
39 }
40
41 [TestMethod]
42 public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRangeException()
43 {
44     // Arrange
45     double beginningBalance = 11.99;
46     double debitAmount = 20.0;
47     BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);
48
49     // Act
50     try
51     {
52         account.Debit(debitAmount);
53     }
54     catch (System.ArgumentOutOfRangeException e)
55     {
56
57         StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);
58         return;
59     }
60
61     Assert.Fail("The expected exception was not thrown.");
62 }
63
64 }
```

Clean Architecture Nedir?

Neden kullanmalıyız? Avantajları nelerdir?

Bu mimarinin yaratıcısı olarak “Uncle Bob” adıyla tanınan Robert C. Martin’dir.

Avantajları

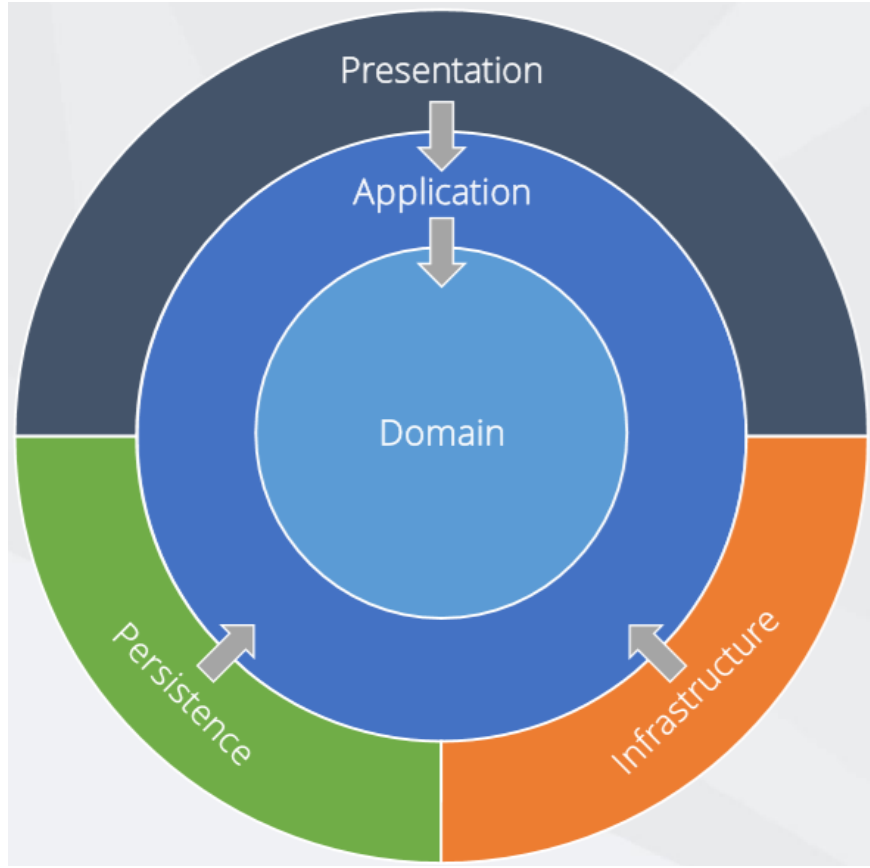
- Framework bağımsız
- Test edilebilir
- Arayüz bağımsız
- Veritabanı bağımsız

Katmanlara neden ayırmalıyız ?

Bu sorunun birden çok yanıtı var aslında. Öncelikle sistemimizin iş parçacıklarının birbirinden ayrı, herhangi bir değişiklikte diğer parçaları da değiştirmeyecek bir yapıda olmasını isteriz. Tabi bu tek sebep değil projenin boyutu arttıkça okunulabilir olması da çok önemlidir. İş parçacıklarını gerçekten birbirinden bağımsız yada bir diğer adıyla gevşek bağıllık(loose-coupling) yapmış isek kolayca test yazabilir ve sonradan yapılan değişikliklerde sistemimizde herhangi bir sıkıntı olup olmadığını gözlemleyebiliriz.

Katmanlar

Aşağıdan da görebileceğiniz gibi katmanlarımız iç içe daireler şeklinde ilerliyor. Her bir daire sadece bir içteki daireye ihtiyaç duyar. Mesela Application katmanının bağımlılığı sadece Domain katmanıdır.



Clean Architecture Yapısı

Domain Katmanı

Domain katmanında projenin kesinlikle ihtiyaç duyacağı olmazsa olmazlarımızdan olan elemanlarımız olacak bunlar nedir ?

Entities

ORM araçlarının gelişmesiyle beraber proje içinde oluşturduğumuz sınıflar ile veritabanı oluşturabilmemiz bu katmanda çok işimize yarıyor.

Value Object

Kendine ait eşsiz bir kimliği olan nesneler Entity olarak adlandırılırken, kendine ait bir kimliği olmayanlar ise value object olarak adlandırılır. Immutable ve mutable kavramındaki mutable: Entity ve immutable: Value Object olarak düşünebiliriz.

Logic : Gerçekten domain ilgilendiren mantıksal işlemlerimiz

Exceptions : Domain için oluşturduğumuz exception sınıflarının da bu katmanda olması gerekiyor.

Application Katmanı

Gelelim Application katmanına, burası bizim genel olarak mantıksal işlemlerimizi yaptığımız katmandır. Örnek olarak gelen isteklerin işlendiği validasyonlarının yapıldığı veritabanı kayıtlarının yapıldığı katmandır. Kafanızda tam oturtturamamış olabilirsiniz ama sorun değil detaylı bir şekilde proje üzerinde göreceğiz. Bu katmanın tek bağılılığı domain katmanı olmalıdır.

Interfaces : Mail servisi veya notification arayüzleri olabilir.

Models : Application katmanında kullanacağınız modeller olabilir.

Logic Commands / Queries : Burası önemli servise gelen isteklerin Request ve Response modellerini, bu servislerin mantıksal işlemlerini ve veritabanı kayıtlarının bulunduğu kısım.

Validators : Gelen isteklerin validasyonları bulunur.

Exceptions : Oluşan hatalar için de kişiselleştirdiğimiz Exception sınıflarımız bulunur.

Gelen isteklerin bir okuma işlemi mi yoksa yazma işlemi mi durumlarına göre farklı modeller ve farklı metotlar ile işlemlerimizi yaparız. Kısacası okunabilir, düzenlenebilir ve test edilebilir bir yapı sunar.

Persistence Katmanı

Bu katman genel olarak DbContext işlemleriyle alakalıdır. Migrationların yönetimi, veritabanı tablolarının configleri için oluşturulmuş Fluent API sınıfları ve default veritabanı değerleri. DbContext bu katmanda olmasına rağmen hala veritabanı bağımlılığımız yoktur. ConnectionString AppSettings içinden erişilecek.

DbContext

Migrations

Configurations

Seeding

Infrastructure Katmanı

Bu katmanda ise sisteme eklenecek external şeyler bulunur. Bu katmana hiçbir katmanın bağılılığı olmamalıdır.

Email / SMS

System Clock

Notification

Presentation Katmanı

Bu katmanda isminden anlaşılacağı üzere sunum katmanı. Bu katmanda mantıksal işlemler olmamalıdır. O iş Application katmanında olmalıdır.

SPA - Angular veya React

Web API

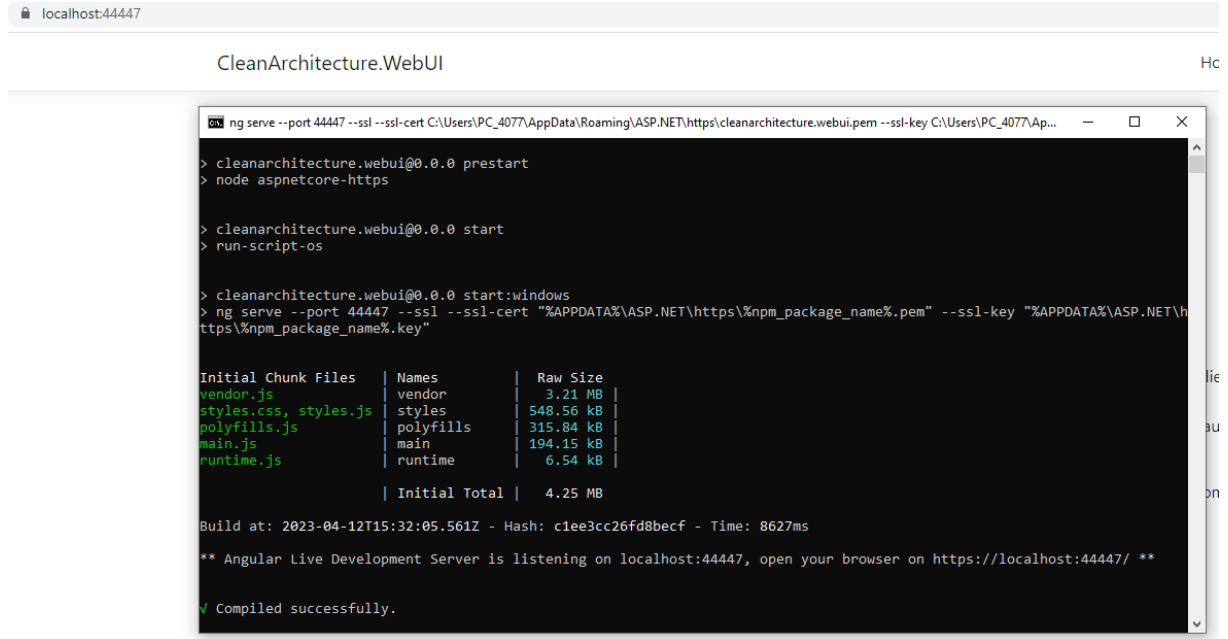
MVC

Web Forms

Clean Architecture

Bu örnek Clean Architecture yapısını açıklayan bir projedir.

.Net ile çalıştırdıktan sonra local hosta bağlanıyoruz. Local host'a bağlandığımızda aşağıdaki görüntüyle karşılaşırız. Burada gerekli JS dosyaları çalıştırır.



```
ng serve --port 44447 --ssl --ssl-cert C:\Users\PC_4077\AppData\Roaming\ASP.NET\https\cleanarchitecture.webui.pem --ssl-key C:\Users\PC_4077\AppData\Roaming\ASP.NET\https\cleanarchitecture.webui.key

> cleanarchitecture.webui@0.0.0 prestart
> node aspnetcore-https

> cleanarchitecture.webui@0.0.0 start
> run-script-os

> cleanarchitecture.webui@0.0.0 start:windows
> ng serve --port 44447 --ssl --ssl-cert "%APPDATA%\ASP.NET\https\npm_package_name%.pem" --ssl-key "%APPDATA%\ASP.NET\https\npm_package_name%.key"

Initial Chunk Files | Names | Raw Size |
vendor.js | vendor | 3.21 MB |
styles.css, styles.js | styles | 548.56 kB |
polyfills.js | polyfills | 315.84 kB |
main.js | main | 194.15 kB |
runtime.js | runtime | 6.54 kB |
| Initial Total | 4.25 MB |

Build at: 2023-04-12T15:32:05.561Z - Hash: c1ee3cc26fd8becf - Time: 8627ms

** Angular Live Development Server is listening on localhost:44447, open your browser on https://localhost:44447/ **

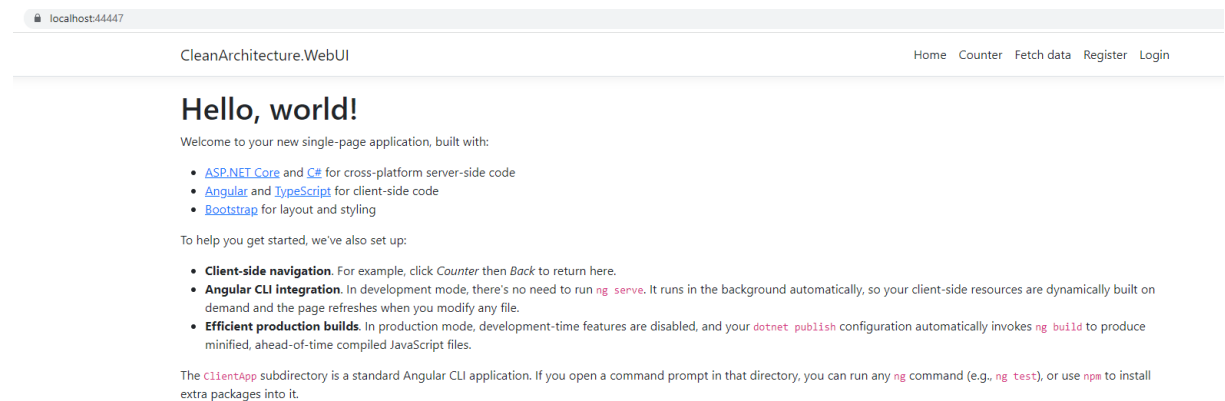
✓ Compiled successfully.
```

Projemizin anasayfası bu şekildedir. Üst menüde ‘Home’, ‘Counter’, ‘Register’, ‘Login’ bulunmaktadır. Bu projede single-page application kullanılmıştır.

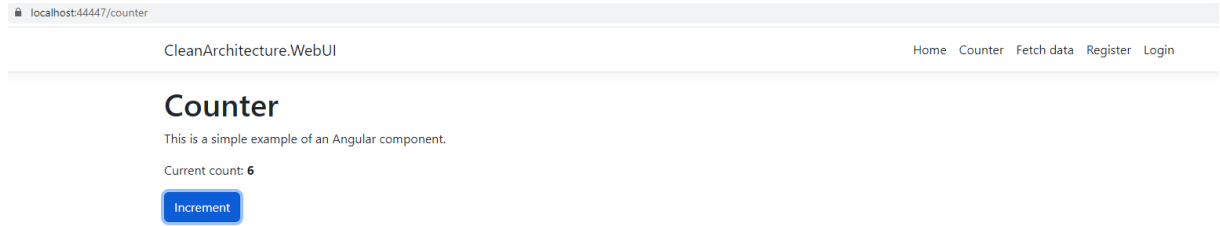
Single Page Application, routing(yönlendirme) işleminin backend tarafından değil de JavaScript tarafından client-side(istemci) tarafında yapıldığı bir web uygulamasıdır. Özetle web sayfamızda yalnızca bir tane index sayfası oluyor ve istemci tarafında sayfalar router ile değişiyor ve kullanıcıya gösteriliyor. Bunu yaparken de web componentlerden yardım alınıyor. Router, ilgili path için sizin belirlediğiniz componentleri kullanıcıya gösteriyor. Bu da yeni bir sayfaya geçmiş etkisi yaratıyor. Yani tüm bu componentlerin değişimi tarayıcınızda oluyor.

Kullanılan teknolojiler :

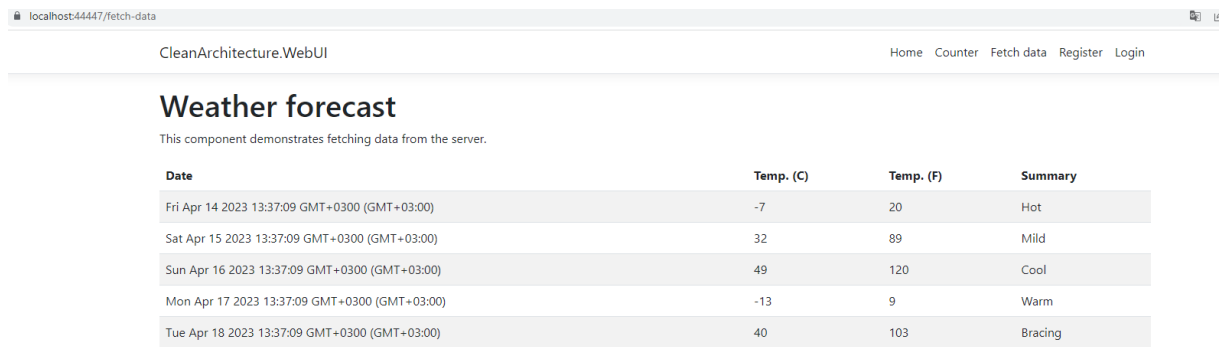
- ASP.NET Core
- C#
- Angular
- TypeScript
- Bootstrap



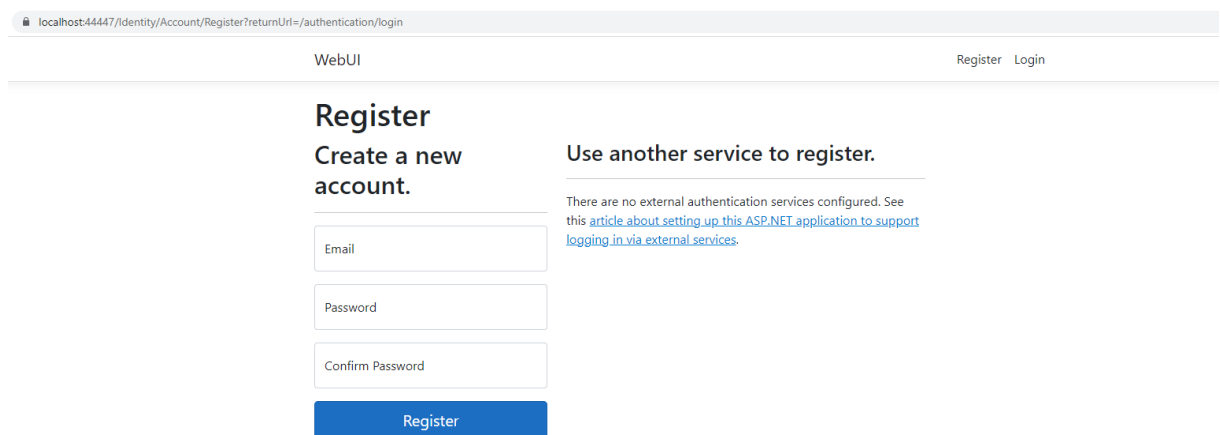
Burada Counter sekmesi görüntülenir.



Bu sekme Fetch data sekmesidir. Bu projede bu sekmeye hava durumu verileri eklenmiştir. Bu eklenen veriler burada gözükmemektedir. İsteğe bağlı olarak bu sekmeye farklı veriler yansıtılabilir.



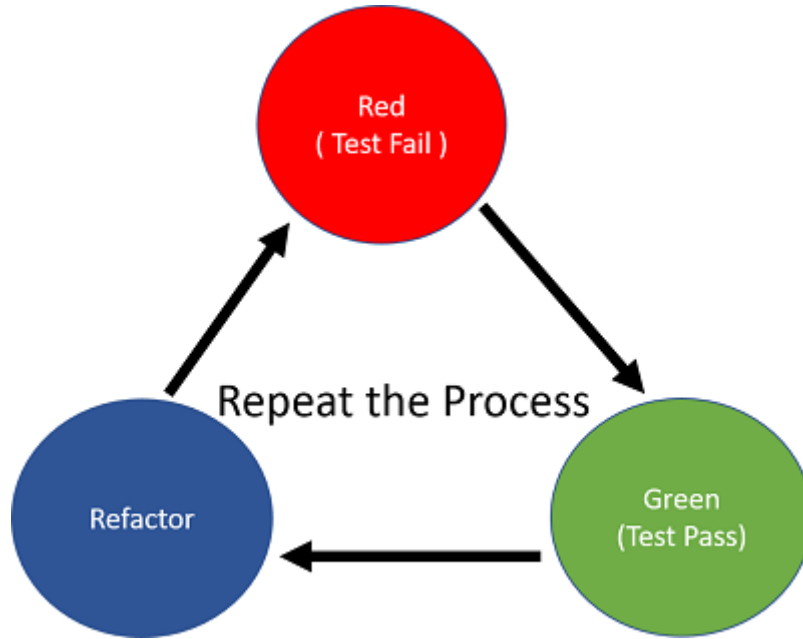
Register sekmesinde kullanıcı yeni bir hesap oluşturur ve sisteme giriş yapar.



C#'ta Birim Testini Kullanarak TDD(Test Driven Development) Uygulaması

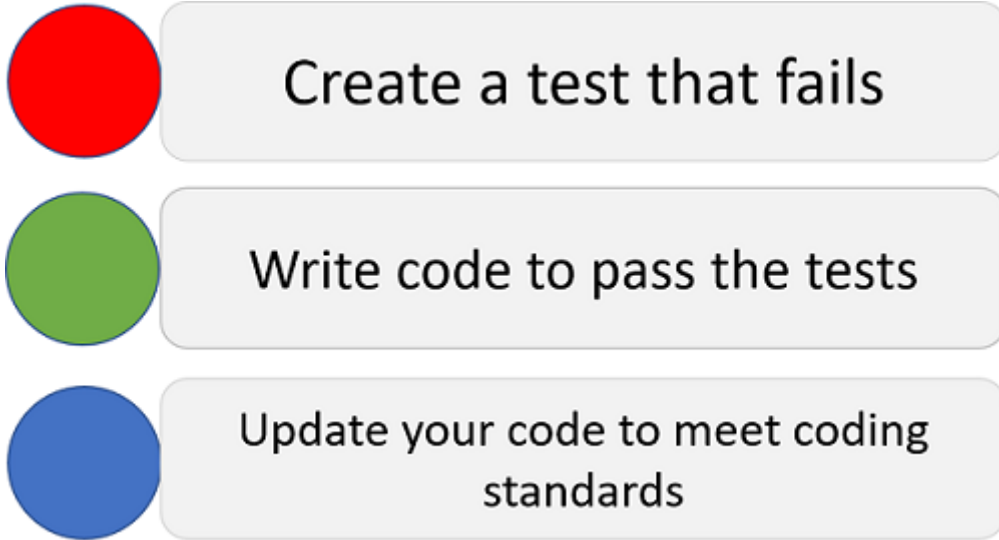
Bu makalede daha önce tartıştığımız gibi, çoğu geliştirici önce Test Yazma, ardından Kod Sonra Yaklaşımı kavramını takip eder, TDD yaklaşımını öğreneceğiz ve nasıl kullanılacağını bileceğiz.

Test güdümlü geliştirme (TDD), temelde her zaman kısa geliştirme döngülerinin tekrarına bağlı olan bir spesifikasyon tekniği ve yazılım geliştirme sürecidir. Test güdümlü geliştirme (TDD) yazılım geliştirme sürecinde, geliştirici önce istenen bir iyileştirmeyi veya yeni işlevi tanımlayan, başlangıçta başarısız olan test senaryosu olarak da adlandırılan otomatik bir test senaryosu yazar, ardından bu testi geçmek için minimum miktarda kod yazar. ve son olarak kabul edilebilir standartlar haline getirmek için yeni kodda refactors kavramını kullanın.



Test odaklı geliştirme (TDD) kullanmanın avantajlarından biri, geliştiricilerin yazılım programları yazarken küçük adımlar atmasını sağlamasıdır. Örneğin, çalışan projelerinize bazı yeni işlevsel kodlar eklediğinizi ve ardından derleyip

test ettiğinizi varsayalım. Testlerinizin, programlarınızın yeni kodunda bulunan kusurlar nedeniyle kırılma olasılığı yüksektir. TDD konseptinin yardımıyla İki yeni kod satırı yazdıktan sonra iki bin satır kod yazdıysanız, bir geliştiricinin bu kusurları bulması ve düzeltmesi çok daha kolaydır. Testleri yeniden derlemeden ve yeniden çalıştırmadan önce birkaç yeni işlevsel kod satırı eklemek genellikle programcılar arasında tercih edilir.



Test Odaklı Geliştirme sloganı Kırmızı, Yeşil, Refactor'dur.

- Kırmızı: Bir test oluşturun ve başarısız olmasını sağlayın.
- Yeşil: Testin gerekli herhangi bir şekilde geçmesini sağlayın.
- Yeniden düzenleme: Projenizdeki tekrarı ortadan kaldırmak ve tasarımı geliştirirken tüm testlerin geçmesini sağlamak için kodu değiştirin.

Kırmızı/Yeşil/Refactor döngüsü, her yeni kod birimi için çok hızlı bir şekilde tekrarlanır.