iOS Unit Testing

Hazırlayan: Adem Akar

Unit Test Nedir ?

**Unit test**, yazılımda en temel **test** çeşitidir. Yazılan fonksiyonların çalışıp çalışmadığının testlerini yapmak için kullanılır. Buradaki en temel noktalardan biri, oluşturulan bütün testler birbirinden bağımsız çalışmaktadır ve her testin tek bir sonucu olmalıdır (true / false)

Unit Test kısaca şöyle tanımlanabilir; bilinen girdilerin, beklenilen değerler ile çıktıların değerlendirildiği ve başka bir kod parçası tarafından çalıştırılan ufak kod parçalarıdır. Bahsettiğimiz bu doğrulama işlemine ise "assertion" denilmektedir. Assertion, birim testin merkezinde bulunur.

Unit Test’lerinin Yararı Nedir?

Unit test kodun test edilen parçalarının doğru olup olmadığının anlaşılmasına yardımcı olur. Kodlarda yaptığınız değişikliklerin, hata tespitlerinin hızlı bir şekilde tespit etmenizde yardımcı olur. Oluşturduğunuz test birimleri sayesinde testlerinizi manuel yapmak zorunda kalmazsınız.

Yazılım birim testleri(unit test) aracılığı ile programın test edilen parçasının doğru olup olmadığının anlaşılmasında yardımcı olur. Kodlarda yaptığınız değişikliklerin, hata tespitlerinin hızlı bir şekilde tespit etmenizde yardımcı olur. Oluşturduğunuz test birimleri sayesinde testlerinizi manuel yapmak zorunda kalmazsınız. Test sınıflarınızı/metotlarınızı yazarken zaman kaybı görüp üşensek de yeni kod parçaları ve özellikleri geldikçe testin size sağladığı zaman tasarufunun etkisini farkedeceksiniz.

Yazacağımız test metotları sayesinde yazdığımız metotların istediğimiz durumlarını yerine getirip getirmediğini kontrol edebiliriz. Test metotları gözümüzden kaçan veya değişiklikler sonrası oluşan hatalarımızı farkedebilir ve bu sayede kodlarımızda yeniden düzenlemeler(refactoring) yapabiliriz.

Test metotlarımızı ayrı bir proje veya test isimli kaynak dizinlerde tutabiliriz. Basit atamaların olduğu metotları belki test etmemeyi tercih edebilirsiniz fakat yazılmış bir proje için test metotları yazıyorsanız önceliğiniz daha önce sık hata aldığınız veya kritik gözüken metotlardan başlamanız oluşabilecek hataları erkenden görmenizde fayda sağlayackatır. Böylece siz veya ekibinizden biri yeni bir özellik eklediğinde test metotları aracılığı ile değişikliğin kodlarınıza etkisini görebilirsiniz.

Yazılan kodun her satırının başka bir kod (test kodu) tarafından otomatik olarak test edilmesini sağlar.

Kodun anlaşılmasını kolaylaştırır.

Daha hızlı yazılım geliştirmeyi sağlar.(Proje başlangıcında projeyi yavaşlatır gibi görünmekte fakat projenin ileri dönemlerinde ciddi bir zaman kazanımı sağlamaktadır)

Koddaki hata oranını azaltır.

Kodların kalitesinin artmasını sağlar.

Hataların çabuk tespit edilip düzenlenmesini sağlar.

Unit testlerde en büyük avantaj yazılımın geliştirilme hızıdır. Unit testlerin olmadığı bir projeyi düşünelim. Bu projede kullanıcı, bir ürün kaydı yapsın. Proje çalıştığında

Kullanıcı önce bilgileriyle giriş yapıyor,

İşlem yapacağı birimi seçiyor,

Kayıt yapacağı kategoriyi seçiyor,

Açılan formda gerekli bilgilerini yazıyor

Kayıt butonuna basıyor

Unit testlerin olmadığı bir projede yazılım geliştirici, yazılan kayıt fonksiyonu hemen arayüz katmanına bağlandığı taktirde runtime’da oluşacak hataların tespiti için bu işlemlerin hepsini tamamlayıp kayıt butonuna bastığında hatayla karşılaşacaktır. Hatayı düzeltip tekrar çalıştıracak, aynı işlemleri tekrar yapacak ve başka bir hatayla karşılaşabilecektir. Bu, yazılımcıya projenin ilerleyen aşamalarında büyük zaman kaybı yaşatmaktadır.

Unit testlerin en büyük avantajı da burada kendini göstermektedir. Bir üst satırda yazdığım örneği unit testlerle gerçekleştiren bir yazılımcı, bu işlemlerin hiçbirini yapmadan kendi oluşturduğu dummy datalarla bu hataları arayüz katmanına bağlama işlemini yapmadan alacak, testlerinin tümü başarıyla çalıştıktan sonra arayüz katmanına bağlama işlemini yapacak ve zaman kaybını önleyecektir.

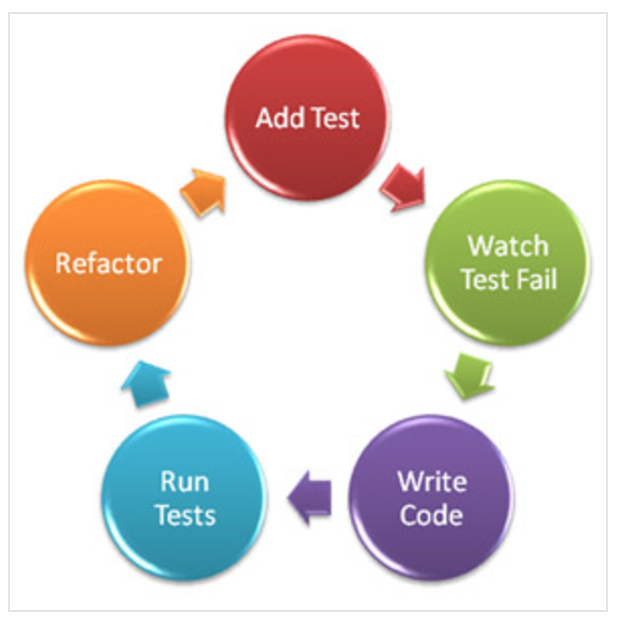
Unit test oluştururken yapılması gereken, sadece kodun çalışacağı değerlerde değil çalışmayacağı değerlerin de parametrelerde gönderilip verilen hataların düzenlenmesidir. Örneğin bir fonksiyon içinde bir yerde bölme işlemi geçiyorsa parametreden bu bölme işleminde böleni 0 yapan bir değer gönderdiğimizde hata vereceğini biliyorsak bu hatanın sonucunda sistemin nasıl bir sonuç üreteceğini de görmemiz gerekmektedir. Bunun için de bir test yazmamız gerekecektir. Classlarımız tamamlandığında testlerinin de oluşturulup classlardaki tüm fonksiyonların testlerini başarıyla tamamlamalarını sağladıktan sonra bir sonraki class’ı kodlamaya başlamak gerekmektedir.

Test Driven Development(TDD)

Kodu yazmadan önce Test Driven Development TDD işlemi uygulanarak test caselerini yazıyoruz ardından bu testleri geçecek kodu yazıyoruz. TDD bu şekilde devam eden bir yazılım geliştirme yöntemidir.

Burada hem caseleri hem de kodu Developer yazmaktadır.

Test Güdümlü Yazılım Geliştirme(TDD)’de test yazmak başlangıçta zaman kaybı olarak görülebilir olsa da aslında durum tam tersidir. Test yazdığınız zaman yazdığınız kodlar %100 test edilmiş olur.

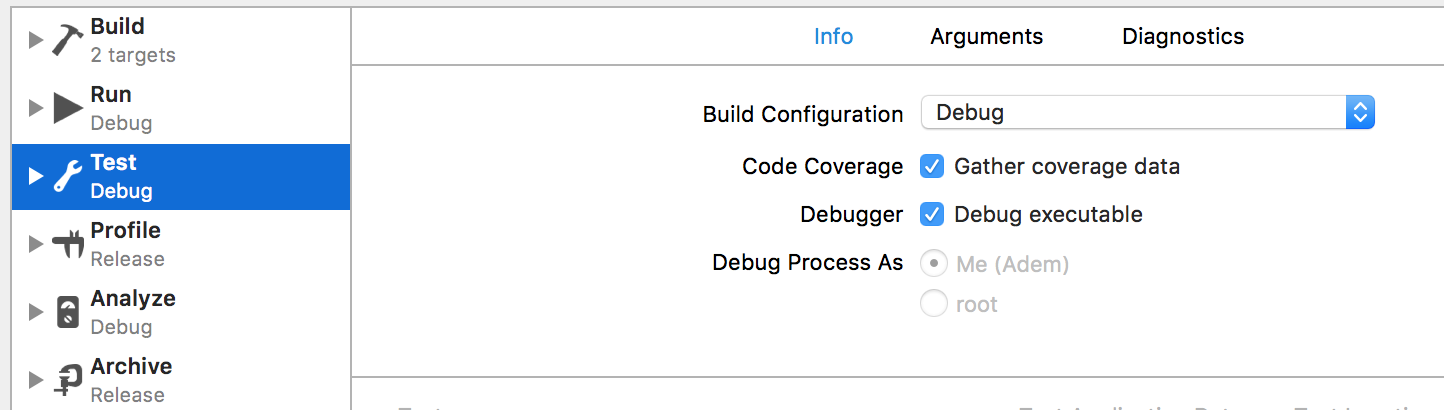


Code Coverage

Code Coverage Test ettiğimiz kodun, yazdığımız tüm koda oranı demektir. Genelde % olarak baz alınır. Code coverage’ımız ne kadar yüksekse unit testlerimizden faydalanma oranımız da o derece yüksektir.

Kodumuzda bir hata olduğunda, hata cover etmediğimiz bir kod bölümüne denk gelmişse biz bunu unit testler ile bulamayacağız demektir, bu yüzden Code Coverage oranının yüksek olması önemlidir.

Projemizde Code Coverage’i Edit Scheme bölümünden aktif edebiliriz.



Unit Testin Test Sürecindeki Yeri

Test sürecinin ilk ayağını Unit Test oluşturur. Adımlar şu şekildedir:

Unit Testing

Integration testing

Functional Interface Testing

Regression Testing

Acceptance Testing

System Testing

Stress Testing

Performance Testing

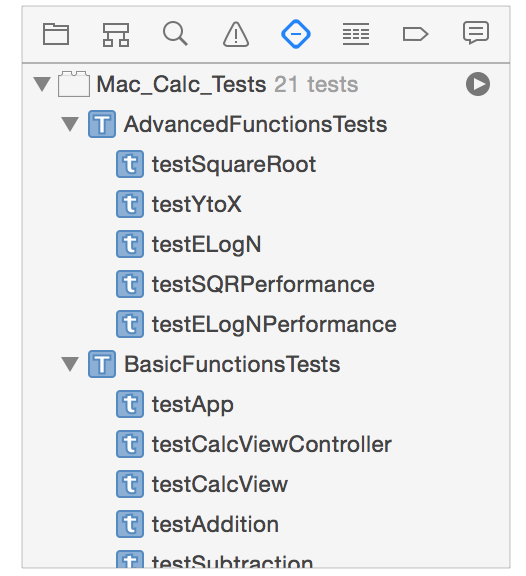
Unit Testing in XCode

XCTest framework’ü ile unit testler, performance testleri, ve UI testler yazabiliriz.

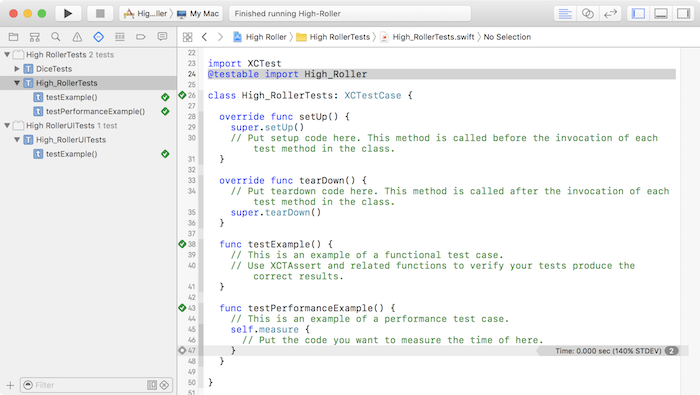
XCTest Xcode 5.0 ve sonraki sürümlerde kullanılabilir. XCTest dışında third party başka Unit Test frameworkleri de kullanılabilir.

XCode Test Navigator

XCode Test Navigator ile yazdığınız testlerin başarılı olup olmadığını kolaylıkla izleyebilirsiniz.



XCTest



Projemize Unit Test eklediğimiz zaman dört aşamalı bir Test Case sınıfı karşımıza çıkar. setUp, teardown, testExample, testPerformanceExample. Şimdi bu bölümleri inceleyelim :

**setUp** : Test Case sınıfındaki her test method çağrılmasından önce çağrılır. Initialize işlemleri burada yapılır.

**tearDown** : Test Case sınıfındaki her test method çağrılmasından sonra çağrılır. setUp ile kullanıma açılan kaynaklar burada serbest bırakılır.

**testExample** : Asıl Test işleminin yapıldığı methoddur. Burada dikkat edilmesi gereken bir nokta test metodları “test”anahtar keyi ile başlamak zorundadır.

**testPerformanceExample**: Performansı ölçme işlemi burada yapılır. Bu medot ile yazdığımız testlerin performansını ölçebiliriz.

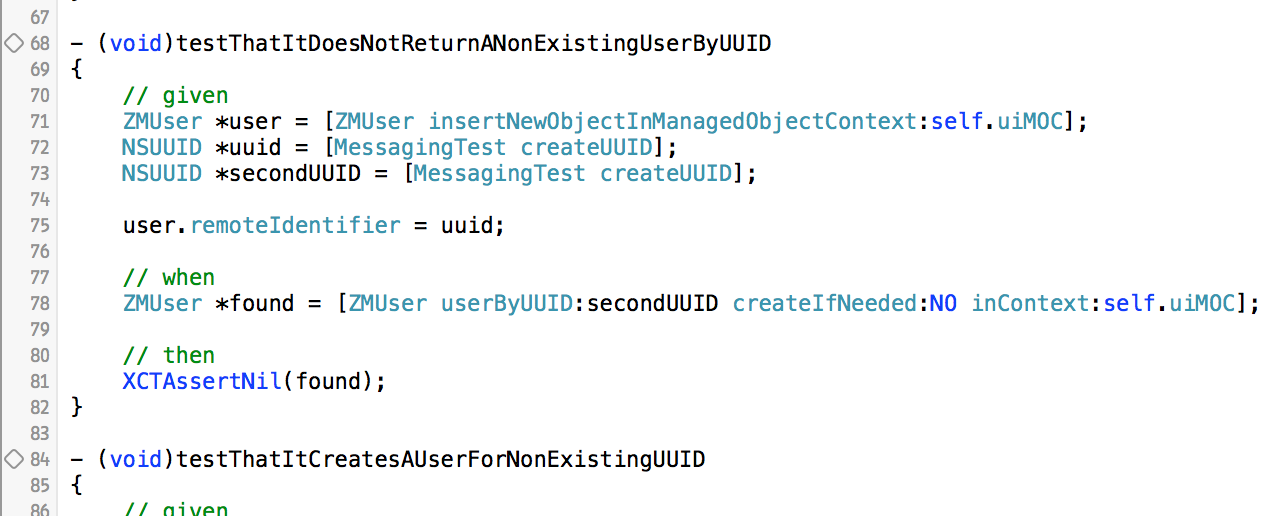
Given-When-Then

Unit Test adımlarını üç bölümde ele alabiliriz. Given-When-Then (GWT)

**Given**: Model nesneleri oluşturarak veya test edilen sistemi belirli bir duruma getirerek test ortamını oluşturur

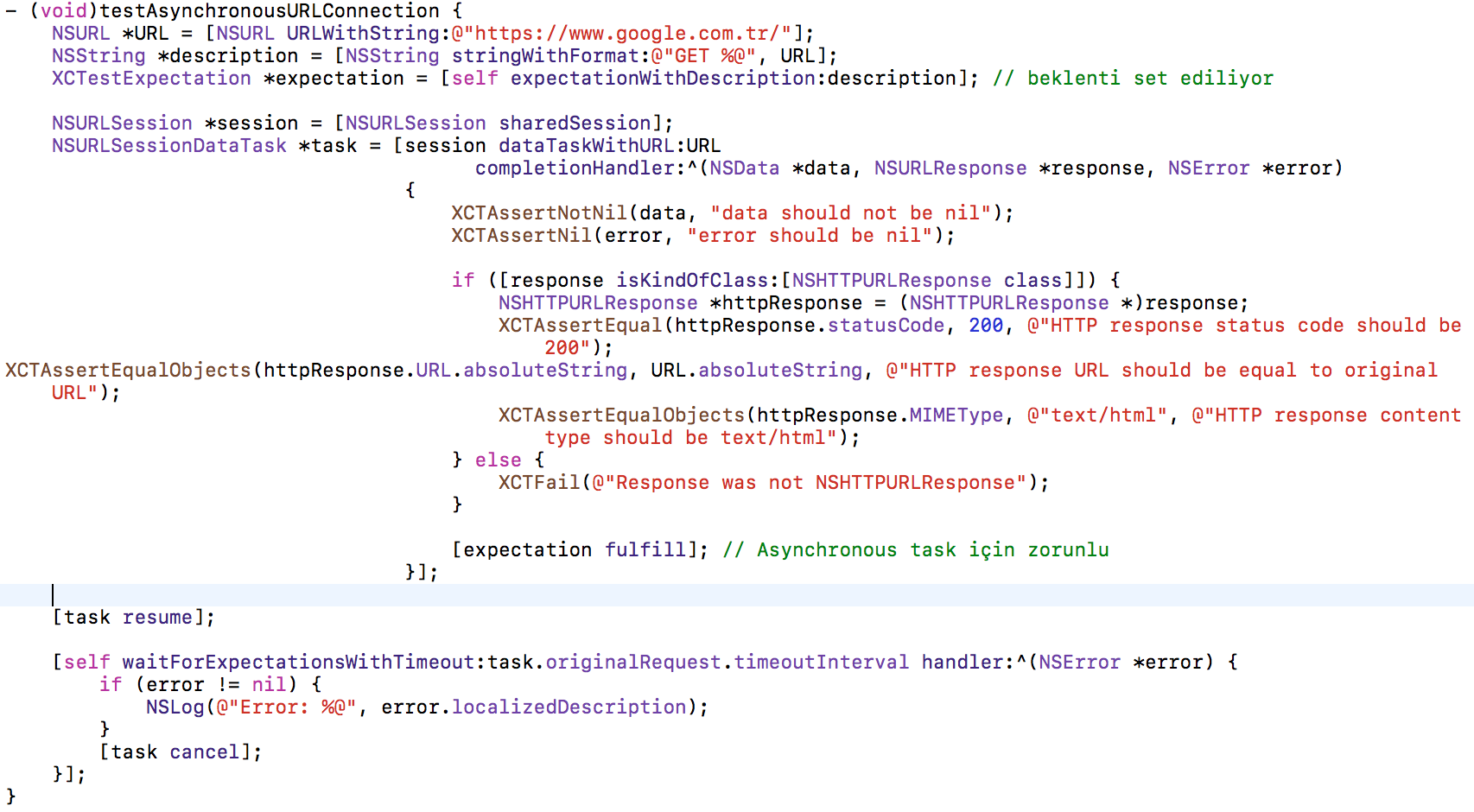
**When**: Test etmek istediğiniz kodu içerir.

**Then**: Sonucumuzu kontrol ediyoruz



Asynchronous Tests

Asynchronous Task’lar, arka planda çalışması gereken zaman alıcı görevler olabilir. Bu durum unit testler için bir problem oluşturur. Dolayısıyla, testlerimizi çalışan görev tamamlandığında çalışacak şekilde düzenlemeliyiz.



Asynchronous işlemler için XCTestExpectation tanımlamak gerekir. Unit test akışına Asynchronous işlemin bittiği bildirmek için fulFill metodu kullanılır. Test akışı boyunca Asynchronous task’tan başka Asynchronous Task ile ilişkili işlemlerimiz de varsa burada waitForExpectationsWithTimeout metodunu kullanmak gerekir. Bu medot Asynchronous taskı verilen timeOut süresi boyunca bekler. Verilen sure içinde task bitmez ise test fail verir.

Unit Text Tutorial

Email adresimizin geçerli bir adres olup olmadığını control eden bir metod yazdığımızı düşünelim(validEmail) , metod aşağıdaki gibi. Bu metodun doğru çalışıp çalışmadığını control eden bir Unit test yazalım.

- (BOOL) validEmail:(NSString\*) emailString {

if([emailString length]==0){

return NO;

}

NSString \*regExPattern = @"[A-Z0-9a-z.\_%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}";

NSRegularExpression \*regEx = [[NSRegularExpression alloc] initWithPattern:regExPattern options:NSRegularExpressionCaseInsensitive error:nil];

NSUInteger regExMatches = [regEx numberOfMatchesInString:emailString options:0 range:NSMakeRange(0, [emailString length])];

NSLog(@"%lu", (unsigned long)regExMatches);

if (regExMatches == 0) {

return NO;

} else {

return YES;

}

}

Unit Test Case sınıfımızda setup içinde gerekli Initialize işlemleri yapıyoruz.

testValidEmail test metodu ile validEMail metodumuzun doğru çalışıp çalışmadığını kontrol ediyoruz. Bu işlem için XCTAssertTrue Unit test metodundan yararlanıyoruz. Test sonucu true ise test başarılı gerçekleşmiş demektir.

@property (nonatomic,strong) ViewController \*vc;

-(void)setUp {

[super setUp];

self.vc=[[ViewController alloc] init];

}

-(void)tearDown {

self.vc=nil;

[super tearDown];

}

-(void)testValidEmail

{

NSString \*originalString = @"adem@hotmail.com";

XCTAssertTrue([self.vc validEmail:originalString]);

}