1903013034/M.Ömer Müştekin

1) İyi bir Birim Testinin dört esası size göre nelerdir?

Birim testler projelerin gelişiminde önemli bir yere sahiptir. Geliştirme sürecine, kişiye ve sonuncunda projeye daha kaliteli katkı sağlamaktadır. Bozukluklarının fark edilmesi ve bu hataların ileriki aşamaya geçmeden önce düzeltilmesini sağlar.

Birim testler yazıldığı sürece hataların tespiti, yeniden düzenleme (refactoring) ve kod bakımı kolaylaştırılmış olur. Birim testler sayesinde yapılan analizler geliştirme sürecinde konularının daha hızlı sonuçlandırılması, regresyon testlerinden başarıyla geçmesi ve daha sağlıklı işleyen fonksiyonlar ortaya çıkararak ileriye dönük büyük faydalar sağlamaktadır.

Bana göre iyi bir birim testin dört ana başlığı şu şekilde olmalıdır:

a) Küçük ve İzole Olmalı (Küçük Kod Parçaları ve Bağımsızlık)

Birim testleri tek başınadır. Birim testler yalıtılmış olarak çalıştırılabilir ve dosya sistemi veya veritabanı gibi dış faktörlere bağımlılığı olmamalıdır.

Testler bir seferde yalnızca bir şeyi test etmelidir. Hepsi bir özelliği/davranışı test ettikleri sürece, birden fazla iddiada sorun yoktur. Bu bağlamda bir test başarısız olduğunda, sorunun yeri tam olarak belirlenmelidir.

Birim testleri yazarken bağımlılıkları izole etme işlemini, bağımlı olunan nesneyi taklit ederek yaparız.

Testlerimizde ne kadar fazla kod parçası test edilirse, testimizin sağlam bir kapsama sahip olması ve başarısız sonuçlanmasının sebebini bulmak zorlaşmaktadır. Bir şey bozulduğunda, bu testler de başarısız olacak ve sorunu hızlı bir şekilde tespit edebileceksiniz. Bu yüzden tüm yöntemi bir kerede test etmemeli ve küçük kod parçalarına sahip olmalıyız.

b) Regresyona Karşı Koruma Sağlamalı

Regresyon hataları, uygulamada bir değişiklik yapıldığında yani refactoring sonrası ortaya çıkan hatalardır. Regresyon hatalarının önüne geçmek için regresyon tesleri yapılır. Her değişiklik ve hata düzeltmesinin ardından yapılan regresyon testleri, sistemin bütünlüğü test eder.

Günümüzde yaygın olarak kullanılan İterative Development sürecinde genellikle her bir döngü için regresyon testi uygulanır. Bu sebeple birim testler regresyon testlerin kategorize edilmiş hali olarak düşünülebilir.

Bu bilgilerden yola çıkarak; birim testi sayesinde her derlemeden sonra veya kod satırını değiştirdikten/refactoring yaptıktan sonra tüm test paketinizi yeniden çalıştırabilirsiniz. Bu sayede yeni kodunuzun mevcut işlevselliği bozulmadığından size güven verir. Sonuç olarak iyi bir birim testi regresyona karşı kod yapınızı güvende tutmalıdır.

c) Daha Az Bağlanmış Bir Kod Yapısına Sahip Olunmalı (Low Cohesion)

Kod yapımız kendi içerisinde sıkı bir şekilde birleştirildiğinde yani yüksek bağlılık (high cohesion) olduğunda birim testi uygulamak daha zor olabilir. Bu sebeple zor olmayan ve iyi bir birim testi için test edilecek kod yapımızın kendi içinde low cohesion olması istenir.

d) Birim Test Hızlı Olmalı, Kısa Tutulmalı ve Çok Fazla Olmamalı

Birim testinin yazılması test edilen kodla karşılaştırıldığında orantısız bir şekilde uzun sürmemelidir. Kodu yazmakla karşılaştırıldığında kodun test edilmesi çok uzun sürüyorsa, yazılan birim testi iyi bir birim test olarak değerlendirmek pek mümkün olmaz. Bu sebeple birim testlerinin çalıştırılması çok az (milisaniye) zaman almalı ve hızlı olmalıdır.

Birim testlerimiz uzun olmaktan ve gereksiz tekrarlar barındırmaktan kaçınmalıdır. Birim testlerde kod yapımızın "Don't Repeat Yourself" ilkesine dayanması iyi bir birim test niteliğine ulaşmamızda büyük katkı sahibi olacaktır. İmkansız senaryoları test etmek ve %100 test kapsamı hedeflemek zaman kaybıdır.

Olgun projelerin binlerce birim testi olması yaygın bir durum değildir. Birim testlerimizin sayısını müşterinin ihtiyacını karşılayacak ve kodumuzu başarılı bir şekilde test edecek düzeyde minimalize etmemiz iyi bir birim testi değerlendirmesinde önemli bir paya sahiptir.

Sonuç

Tüm bunların sonucunda iyi bir birim testin aslında ne kadar çok koşula dayandığını ve tüm bu koşulların bir arada bir bütün olarak bulunmasıyla birlikte bir birim testin "iyi" olarak değerlendirilebileceğini anlıyoruz.

Martin Fowler'ın yüksek seviyeli testlerin bile birim testlerde yapılan yanlışlık veya eksiklik neticesinde başarısız olabileceğini söylediği sözleriyle bu kısmı sonlandırmak istiyorum. Bu sözlerde geçen birim testteki "yanlışlık veya eksiklikler" durumlarının da bu soruda dört başlıkta değerlendirdiğim birim testlerin özelliklerinden bir veya birkaçının noksanlığı neticesinde ortaya çıkabileceğini de belirtmek isterim.

"I always argue that high-level tests are there as a second line of test defense. If you get a failure in a high level test...you also have a missing or incorrect unit test."

-Martin Fowler

Kaynaklar

Linkedin

www.Softwaretestinghelp.com

www.learn.microsoft.com

www.mis.sadievrenseker.com

2) Martin Fowler, Unit Testing ve Clean Code İlişkisinin Değerlendirmesi

Bu kısmı alıntıların ardından kaynağını belirttiğim Martin Fowler'ın yazıları ve söylemleri doğrultusunda ve önceki soruda vermiş olduğum cevaplarla bağlantılı bir şekilde kısımlara ayırarak değerlendireceğim. Fakat yine burada bu kısımların yalnızca karmaşıklığı gidermek ve okunabilirliği arttırmak için var olduğunu, birbirlerinden ayrı olmadığını ve tüm bu yazılanların aslında bir bütün olarak değerlendirilmesi gerektiğini de belirtmek isterim.

Kısım 1

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

(Herhangi biri, bir bilgisayarın anlayabileceği bir kod yazabilir. İyi programcılar, insanların anlayabileceği kodlar yazar.)

"I find that writing unit tests actually increases my programming speed."

(Birim testleri yazmanın aslında programlama hızımı artırdığını buldum.)

Biz yazılım mühendisleri veya 'iyi' programcılar yazdığımız kodun diğer kişiler tarafından anlaşılabilir ve okunabilir olmasına özen göstermeliyiz. Aksi halde zaten mühendis olmanın veya iyi olarak değerlendirilmenin getirdiği niteliklerden kendimizi mahrum bırakmış ve yalnızca bir programcı olmuş oluruz. Olayın aslında yalnızca birkaç satır kod yazmaktan ibaret olmadığını, ortada bir problemin çözüme kavuşması gerektiğini ve bu çözüme ulaşan yolda birçok metot ve bu yolun aslında bir süreç olduğunu, sürecin sonunda problemin çözüme kavuşmasını bekleyen bir müşteri ve bu müşterinin gereksinimlerini karşılayabilmemiz gerektiğini ve bu süreci maksimum verim ve minimum süre ile tamamlamamızın bizlerden beklendiğini biliriz.

Martin Fowler'ın yukarıda belirttiğim sözleri ve az önce bahsettiğim tüm bu beklentiler ve süreç doğrultusunda bir değerlendirme yapacak olursak; kod yapımızı clean code prensibi ile oluşturmalı ve test kısmında bu prensibin birim testlerin işini kolaylaştırdığını ve gecikmeyi minimuma indirerek süreci sağlıklı bir şekilde test edip sonlandırmamızı sağladığının farkında olmamız gerekir.

Kısım 2

"I'd always been inclined to clean code, but I'd never considered it to be that important."

(Her zaman kodu temizlemeye meyilliydim, ancak bunun hiç bu kadar önemli olduğunu düşünmemiştim.)

From Book of Refactoring > Where Did Refactoring Come From?

"I've not succeeded in pinning down the real birth of the term refactoring. Good programmers certainly have spent at least some time cleaning up their code. They do this because they have learned that clean code is easier to change than complex and messy code, and good programmers know that they rarely write clean code the first time around."

(Yeniden düzenleme teriminin gerçek doğuşunu belirlemeyi başaramadım. İyi programcılar kesinlikle kodlarını temizlemek için en azından biraz zaman harcamışlardır. Bunu yaparlar çünkü temiz kodu değiştirmenin karmaşık ve dağınık koddan daha kolay olduğunu öğrenmişlerdir ve iyi programcılar ilk seferde nadiren temiz kod yazdıklarını bilirler.)

From Book of Refactoring > Where Did Refactoring Come From?

Kodumuzu olabildiğince anlaşılabilir ve clean code prensibine uygun kılmak için ilk önce yazdığımız kodun ilk halinin genellikle dağınık ve bu prensibe uygun bir halde olmadığını ve temiz yazılan bir kodu refactor edebilmenin daha kolay olacağını bilmemiz önemlidir. Bu doğrultuda kod satırlarımız oluşmaya ve çoğalmaya başladıkça belki de birden çok refactoring aşamasından geçer ve en nihayetinde kod yapımız bu prensibe uygun hale gelmiş olur. Clean code prensibine uygun hale gelmiş kod parçalarımıza birim testlerimizi daha hızlı, kolay ve herhangi bir hatada problemi açıkça görebileceğimiz bir şekilde uygulayabilmiş oluruz. Tabii ki burada ilk soruya verdiğim cevaplarda değinmiş olduğum, küçük ve bağımsız kod parçalarına sahip olmamız da önemli bir role sahip olacaktır.

Kısım 3

“I've been asked, "Is refactoring just cleaning up code?" In a way the answer is yes, but I think refactoring goes further because it provides a technique for cleaning up code in a more efficient and controlled manner. Since I've been using refactoring, I've noticed that I clean code far more effectively than I did before. This is because I know which refactorings to use, I know how to use them in a manner that minimizes bugs, and I test at every possible opportunity.”

(Bana "Yeniden düzenleme sadece kodu temizlemek mi?" diye soruldu. Bir bakıma cevap evet, ancak yeniden düzenlemenin daha da ileri gittiğini düşünüyorum çünkü kodu daha verimli ve kontrollü bir şekilde temizlemek için bir teknik sağlıyor. Yeniden düzenlemeyi kullandığımdan beri, kodu daha önce yaptığımdan çok daha etkili bir şekilde temizlediğimi fark ettim. Bunun nedeni, hangi yeniden düzenlemeleri kullanacağımı, bunları hataları en aza indirecek şekilde nasıl kullanacağımı ve mümkün olan her fırsatta test etmemdir.)

From Book of Refactoring > 2. Principles in Refactoring > Where Did Refactoring Come From?

Clean code yalnızca bir prensip değil, aynı zamanda bir tekniktir. Bu tekniğin faydaları arasında Martin Fowler’ın yukarıdaki ilk cümlesinin son kısmında ifade ettiği gibi, kodu her defasında test ederek hataları minimalize etmemize kolaylık sunması da bulunuyor. Nitekim bu ifade ile test ve clean code kavramlarını Martin Fowler’ın sözleri ile ilişkilendirmiş oluyoruz.

Kısım 4

"What makes a clean test? Three things. Readability, readability, and readability. Readability is perhaps even more important in unit tests than it is in production code. What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. In a test you want to say a lot with as few expressions as possible."

(Bir testi temiz yapan şey nedir? Üç şey. Okunabilirlik, okunabilirlik ve okunabilirlik. Okunabilirlik, birim testlerinde üretim kodunda olduğundan daha da önemlidir. Testleri okunabilir kılan nedir? Tüm kodu okunabilir kılan aynı şey: netlik, basitlik ve ifade yoğunluğu. Bir testte mümkün olduğunca az ifadeyle çok şey söylemek istersiniz.)

From Book of Clean Code > Chp. 9 Unit Tests > Clean Tests

Birim testi oluşturan kod yapımızın da teste tabii tutulacak kod yapımız gibi Clean Code prensibi dikkate alınarak oluşturulması gerekir. Hatırlarsanız ilk soruda vermiş olduğum son cevaplarda bir birim testin "Don't Repeat Yourself" ilkesine dayanması gerektiğini söylemiştim. Birim testimiz büyük oranda bu ilke neticesinde temiz bir kod yapısına sahip olur.

Yukarıda Robert J. Martin'in Clean Code kitabından alıntılamış olduğum yazısının bu kod yapımızı ve aynı zamanda testimizi temiz (clean) kılan şeyin okunabilirlik olduğu vurgusunu yaptığını görmekteyiz. Bu okunabilirlik, kodumuzun ifade yoğunluğunun güçlü, net ve olabildiğince basit bir dil kullanarak oluşturulması ile doğrudan alakalıdır.

Kısım 5

Şimdiye kadar değindiğimiz konular kapsamında ve tabii daha öncelikli olarak bu soruda cevaplamaya çalıştığım unit test ve clean code ilişkisine ve diğer çoğu cevabıma bir örnek teşkil etmesi açısından, Robert J. Martin'in Clean Code kitabında anlattığı ve birebir yaşamış olduğu gerçek hayat probleminden ve bunun neticesinde çıkarmış olduğu çıkarımlardan bahsettiği yazısını paylaşmak istiyorum:

"Some years back I was asked to coach a team who had explicitly decided that their test code should not be maintained to the same standards of quality as their production code. They gave each other license to break the rules in their unit tests. "Quick and dirty" was the watchword. Their variables did not have to be well named, their test functions did not need to be short and descriptive. Their test code did not need to be well designed and thoughtfully partitioned. So long as the test code worked, and so long as it covered the production code, it was good enough. Some of you reading this might sympathize with that decision. Perhaps, long in the past, you wrote tests of the kind that I wrote for that Timer class. It's a huge step from writing that kind of throw-away test, to writing a suite of automated unit tests. So, like the team I was coaching, you might decide that having dirty tests is better than having no tests. What this team did not realize was that having dirty tests is equivalent to, if not worse than, having no tests. The problem is that tests must change as the production code evolves. The dirtier the tests, the harder they are to change. The more tangled the test code, the more likely it is that you will spend more time cramming new tests into the suite than it takes to write the new production code. As you modify the production code, old tests start to fail, and the mess in the test code makes it hard to get those tests to pass again. So the tests become viewed as an

ever-increasing liability. From release to release the cost of maintaining my team's test suite rose. Eventually it became the single biggest complaint among the developers. When managers asked why their estimates were getting so large, the developers blamed the tests. In the end they were forced to discard the test suite entirely. But, without a test suite they lost the ability to make sure that changes to their code base worked as expected. Without a test suite they could not ensure that changes to one part of their system did not break other parts of their system. So their defect rate began to rise. As the number of unintended defects rose, they started to fear making changes. They stopped cleaning their production code because they feared the changes would do more harm than good. Their production code began to rot. In the end they were left with no tests, tangled and bug-riddled production code, frustrated customers, and the feeling that their testing effort had failed them. In a way they were right. Their testing effort had failed them. But it was their decision to allow the tests to be messy that was the seed of that failure. Had they kept their tests clean, their testing effort would not have failed. I can say this with some certainty because I have participated in, and coached, many teams who have been successful with clean unit tests. The moral of the story is simple: Test code is just as important as production code. It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code."

(Birkaç yıl önce, test kodlarının üretim kodlarıyla aynı kalite standartlarında tutulmaması gerektiğine açıkça karar vermiş bir ekibe koçluk yapmam istendi. Birim testlerinde birbirlerine kuralları çiğneme yetkisi verdiler. "Hızlı ve kirli" parolaydı. Değişkenlerinin iyi adlandırılması gerekmiyordu, test fonksiyonlarının kısa ve açıklayıcı olması gerekmiyordu. Test kodlarının iyi tasarlanmış ve düşünceli bir şekilde bölümlere ayrılmış olması gerekmiyordu. Test kodu çalıştığı ve üretim kodunu kapsadığı sürece yeterince iyiydi. Bunu okuyan bazılarınız bu karara sempati duyabilir. Belki de uzun zaman önce o Timer sınıfı için yazdığım türden testler yazdınız. Bu tür bir atılan test yazmaktan bir dizi otomatik birim testi yazmaya kadar büyük bir adım. Yani, koçluk yaptığım takım gibi, kirli testler yaptırmanın hiç test yapmamaktan daha iyi olduğuna karar verebilirsiniz. Bu ekibin anlamadığı şey, kirli testlere sahip olmanın, test yaptırmamaktan daha kötü olmasa da eşdeğer olduğuydu. Sorun, üretim kodu geliştikçe testlerin değişmesi gerektiğidir. Testler ne kadar kirliyse, değiştirmek o kadar zor olur. Test kodu ne kadar karışıksa, yeni üretim kodunu yazmak için harcadığınız zamandan daha fazla yeni testleri pakete tıkmak için harcamanız o kadar olasıdır. Üretim kodunu değiştirdikçe, eski testler başarısız olmaya başlar ve test kodundaki karışıklık, bu testlerin tekrar geçmesini zorlaştırır. Böylece testler giderek artan bir sorumluluk olarak görülüyor. Sürümden sürüme, ekibimin test paketini korumanın maliyeti yükseldi. Sonunda geliştiriciler arasında en büyük şikayet haline geldi. Yöneticiler, tahminlerinin neden bu kadar büyüdüğünü sorduğunda, geliştiriciler testleri suçladı. Sonunda test takımını tamamen atmak zorunda kaldılar. Ancak, bir test takımı olmadan, kod tabanlarındaki değişikliklerin beklendiği gibi çalıştığından emin olma yeteneklerini kaybettiler. Bir test takımı olmadan, sistemlerinin bir kısmında yapılan değişikliklerin, sistemlerinin diğer kısımlarını bozmamasını sağlayamazlardı. Böylece kusur oranları yükselmeye başladı. İstenmeyen kusurların sayısı arttıkça, değişiklik yapmaktan korkmaya başladılar. Değişikliklerin yarardan çok zarar getireceğinden korktukları için üretim kodlarını temizlemeyi bıraktılar. Üretim kodları çürümeye başladı. Sonunda hiçbir test, karışık ve hatalarla dolu üretim kodları, hayal kırıklığına uğramış müşteriler ve test çabalarının onları

başarısızlığa uğrattığı duygusuyla baş başa kaldılar. Bir bakıma haklılardı. Test çabaları onları başarısızlığa uğratmıştı. Ancak, bu başarısızlığın tohumu, testlerin dağınık olmasına izin verme kararlarıydı. Testlerini temiz tutsalardı, test çabaları başarısız olmazdı. Bunu kesin olarak söyleyebilirim çünkü temiz birim testlerinde başarılı olan birçok takıma katıldım ve koçluk yaptım. Hikayenin ahlaki basit: Test kodu, üretim kodu kadar önemlidir. İkinci sınıf vatandaş değildir. Düşünce, tasarım ve özen gerektirir. Üretim kodu kadar temiz tutulmalıdır.)

From Book of Clean Code > Chp. 9 Unit Tests > Keeping Tests Clean

Kaynaklar

www.martinfowler.com

Book of “Refactoring: Improving the Design of Existing Code” written by Martin Fowler

Book of “Clean Code” written by Robert C. Martin

3) Unit Test Frameworks ve çalıştıkları ortamları araştırdıktan bunlarla ilgili olarak size göre nasıl bir sınıflandırma yaparsınız?

Unit testleri yazmak ve çalıştırmak için unit test frameworkleri/araçları kullanılır.

Daha kolay ve esnek taklit nesneler yaratabilmek için mocking framework kullanılır. Bu framework aracılığı ile bağımlı olunan nesnenin taklidinin nasıl davranması gerektiğini belirterek istediğimiz şekilde taklit bir nesne yaratıp kullanabiliriz. .Net içerisinde en sık kullanılan mocking framework Moq'dur. Moq haricinde NSubstitute, FakeItEasy gibi alternatifler de mevcuttur.

Integration testlerin yazılması ve çalıştırılması için kullanılan araç/frameworkler genellikle unit testler ile aynı araçlardır. Integration testlere özgü ayrı bir araç/framework kullanılmasına genellikle ihtiyaç olmamaktadır.

Functional testlerin yazılması ve çalıştırılması için, unit ve integration testler için kullanılan araçlara ek olarak farklı araç/frameworkler kullanılması gerekir. Kullanılacak ek araçlar uygulamanın tipine göre değişiklik gösterebilir. UI sahibi bir uygulama için UI test araçları kullanılırken, webapi için yalnızca TestServer/Host araçlarının kullanılması gerekebilir.

Tüm bunlar neticesinde birim test frameworklerini şu şekilde sınıflandırabilirim:

a) Hazır olarak bir proglamlama dilinin içinde bulunan ve test araçlarının içinde barındırıldığı frameworkler.

b) Ayrı ve dışarıdan erişilebilen, her programlama dili için özel olarak geliştirilmiş frameworkler.

c) Uygulamanın yapısı ve tipine göre kullanılan frameworkler.

d) Sahte nesneler yaratmak gibi çeşitli amaçlarla kullanılan frameworkler.

Kaynaklar

www.learn.microsoft.com

www.Wikipedia.org

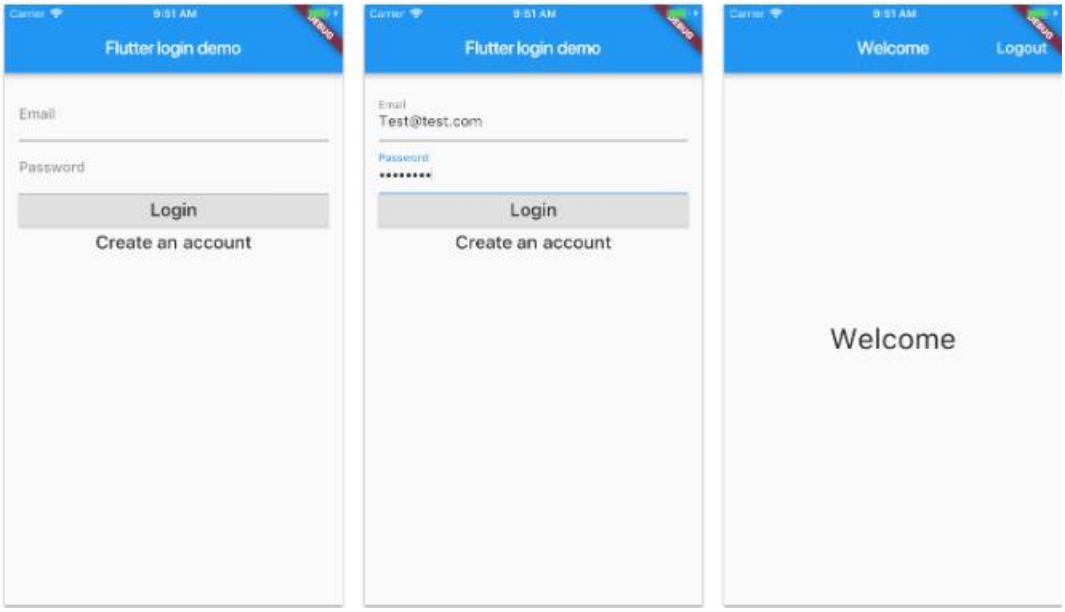
www.Medium.com

4) Bir kod parçasının birim testini 3A yaklaşımı ile yazın, gerekli açıklamaları yaparak ekran çıktılarını paylaşın.

Bu kısımda ilk olarak yalnızca birim testler ile, ardından da birim testleri 3A yöntemi kullanarak uygulayacağım iki adet senaryo hazırladım. Böylece 3A yöntemini daha iyi anlayabileceğimizi düşünüyorum.

A) Birim Test Yazıp Uygulamak

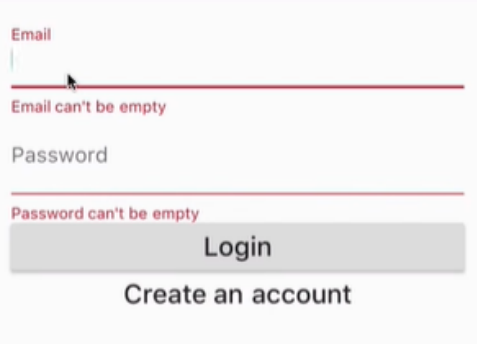
Arayüz



Kullanıcı beklenen durumda e-mail ve password bilgilerini girerek login olma işlemini gerçekleştirir.

Problem

Kullanıcı beklenmeyen durumda login olma işlemi için girilmesi gerekli olan e-mail ve password bilgilerini boş bırakıp login olmaya çalıştığında, kullanıcıya aşağıdaki gibi bir uyarı verilmesi gerekmektedir.



Class ve Fonksiyonlar

```
List<Widget> buildInputs() {
  return [
    TextFormField(
      decoration: InputDecoration(labelText: 'Email'),
      validator: EmailFieldValidator.validate,
      onSaved: (value) => _email = value,
    ), // TextFormField
    TextFormField(
      decoration: InputDecoration(labelText: 'Password'),
      obscureText: true,
      validator: PasswordFieldValidator.validate,
      onSaved: (value) => _password = value,
    ), // TextFormField
  ];
}
```

```
class EmailFieldValidator {
  static String validate(String value) {
    return value.isEmpty ? 'Email can\'t be empty' : null;
  }
}

class PasswordFieldValidator {
  static String validate(String value) {
    return value.isEmpty ? 'Password can\'t be empty' : null;
  }
}
```

EmailFieldValidator ve *PasswordFieldValidato*r adlı iki adet classım ve bu classların içinde kullanıcının girmiş olduğu e-mail ve password bilgilerinin boş olup olmadığını kontrol eden ve buna karşılık ekrana hata mesajı yazdıran bir fonksiyon mevcut.

Birim Test Kodu

```
void main() {

  test('empty email returns error string', () {

    var result = EmailFieldValidator.validate('');
    expect(result, 'Email can\'t be empty');
  });

  test('non-empty email returns null', () {

    var result = EmailFieldValidator.validate('email');
    expect(result, null);
  });

  test('empty password returns error string', () {

    var result = PasswordFieldValidator.validate('');
    expect(result, 'Password can\'t be empty');
  });

  test('non-empty password returns null', () {

    var result = PasswordFieldValidator.validate('email');
    expect(result, null);
  });
```

Her bir durum için kodumun doğru çalışıp çalışmadığını birim testler uygulayarak kontrol ettim.

B) 3A Yöntemi ile Birim Test Yazıp Uygulamak

Problem

Bir yapılacaklar listesi sınıfıma ait olan ve en önemli özelliklerden biri olan listeye eleman ekleme işleminin düzgün çalışıp çalışmadığını test etmemiz gerekir.

Kod

TodoViewModel adında bir sınıfım var ve bu sınıf kendi içerisinde listeye eleman ekleyen *addItemToList* adında bir fonksiyona sahip.

```
class TodoViewModel extends ChangeNotifier {
  List < TodoItem > todos = [];

  void addItemToList(TodoItem item) {
    todos.add(item);
    notifyListeners();
  }
}
```

Öğelerin listeye başarıyla eklenip eklenmediğini kontrol etmek için 3A yöntemi ile birim testimi uyguluyorum:

```
test('Should get all items added to the list', () {
  // Arrange
  final todoViewModel = TodoViewModel();
  final item = TodoItem(
    id: 1,
    title: 'Buy groceries',
    description: 'Go to the mall and shop for next month's
stock.',
    createdAt: 1,
    updatedAt: 1,
  );

  // Act
  todoViewModel.addItemToList(item);
  todoViewModel.addItemToList(item);
  todoViewModel.addItemToList(item);

  // Assert
  expect(todoViewModel.todos.length, 3); });
```

Bu test senaryosunda Arrange-Act-Assert modelinin kullanımını özetlemem gerekirse;

1. Arrange: Bu adım, bir *ChangeNotifier* alt sınıfı olan *TodoViewModel* ve *TodoItem* nesnesinin örneğini oluşturur.

2. Act: *TodoItem* nesnesinin aynı örneğini ekleyen *addItemToList* yöntemini çağırarak *TodoViewMode*l'e etki eder.

3. Assert: *addItemToList* yöntemini üç kez çağırmaya dayalı olarak yapılacak işlerin üç öğeye sahip olduğunu doğrular.

Bu şekilde 3A yöntemini kullanarak listeye eleman eklenip eklenmediğini test etmiş olduk.

Kaynaklar

https://docs.flutter.dev/testing

https://codelabs.developers.google.com

https://developer.android.com