FLUTTER UYGULAMANIZ DOSYALARLA ÇALIŞABİLİR

Bu noktaya kadar, bir Flutter uygulaması oluşturmayı, uygulamanın her türlü cihaz ve yönde nasıl görüneceğini ve nasıl düzenleneceğini tam olarak kontrol etmeyi öğrendin. Bu oldukça havalı! Ayrıca, Form alanlarıyla verileri nasıl koruyacağını da biliyorsun. Peki, bu verileri nasıl kaydedeceğiz? İlk etapta bu verileri nasıl okuyacağız?

Uygulamanın verileri yalnızca iki kaynaktan gelebilir: Cihazın kendisinden veya harici bir sunucu ile veri alışverişi yaparak. Harici sunucularla veri alışverişini bir sonraki bölümde ele alacağız. Bu bölümde ise, cihaz içindeki depolama seçeneklerinden veri okuma ve yazmayı öğreneceğiz. Ancak bölümü tamamlayıp bu noktaya ulaşabilmemiz için önce belirli ön bilgilerde ustalaşmamız gerekiyor. Bunlar arasında JSON ile çalışmak ve eşzamansız işlemleri yönetmek yer alıyor. İşte bu bölüm için planımız:

* Flutter uygulamanıza kütüphaneler ekleme
* Uygulamanıza dosyaları dahil etme
* Futures, async ve await kullanımı
* Yerel bir dosya okuma/yazma
* JSON dönüştürme
* Kullanıcı tercihlerini kaydetme

EŞZAMANSIZ İŞLEMLER NEDİR?

Eşzamansız işlemler (asenkron işlemler), programın bir işlemi tamamlamasını beklemeden çalışmaya devam etmesini sağlayan işlemlerdir.

Normalde, bir işlem sırasıyla ilerler ve bir adım tamamlanmadan diğerine geçmez. Buna **eşzamanlı (senkron) işlemler** denir. Ancak bazı işlemler zaman alabilir, örneğin:

* Bir sunucudan veri çekmek
* Dosya okuma veya yazma
* Kullanıcının giriş yapmasını beklemek

Bu tür işlemler, uygulamanın donmasına veya yavaş çalışmasına neden olabilir.

İşte bu noktada **eşzamansız (asenkron) işlemler** devreye girer. Flutter’da Future ve async-await gibi yapılar kullanarak bu işlemleri arka planda çalıştırabilir ve uygulamanın akıcı kalmasını sağlayabiliriz.

Flutter uygulamanıza kütüphaneler ekleme

Birçok yetenekli geliştirici, senin ve benim Flutter uygulamalarımızda kullanabileceğimiz harika araçlar yazdı. Bunlara genellikle **kütüphane (library)** denir ve yapmamız gereken tek şey:

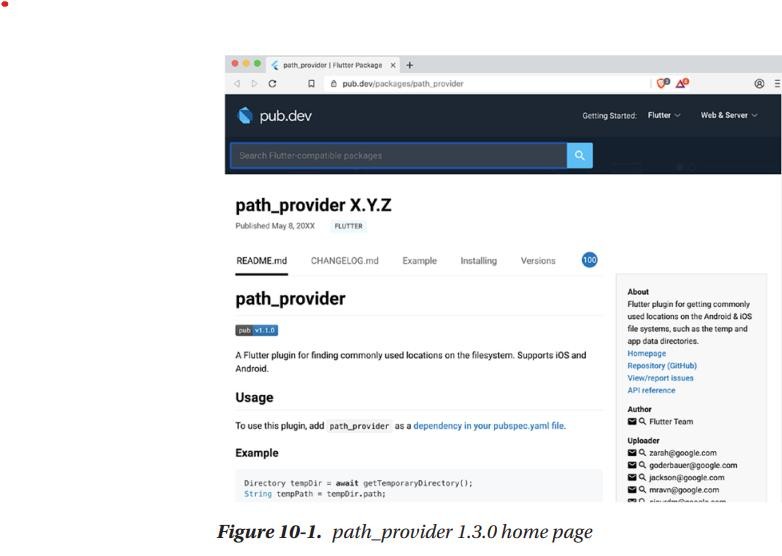
1. Beğendiğimiz bir kütüphane bulmak,
2. pubspec.yaml dosyamıza eklemek,
3. Bir Dart dosyasında içe aktarmak (import etmek),
4. Kodumuzda kullanmaktır.

Bir kütüphane bulma:

Açıkçası, bu en zor kısım çünkü arama yapabileceğimiz tek bir yer yok. Kulağa basit gelse de en iyi yöntem **Google’da arama yapmaktır**.

Bu bölümde dosya okuma ve yazma işlemlerini öğreneceğimiz için bunu bir örnek olarak ele alalım. **"Flutter read file"** (Flutter dosya okuma) şeklinde Google’da arama yaparsak, ilk beş sonucun hepsinin **path\_provider** adlı bir kütüphaneyi önerdiğini görürüz.

Bunun üzerine **"Flutter path\_provider"** şeklinde arama yaptığımızda, [**https://pub.dev/packages/path\_provider**](https://pub.dev/packages/path_provider) adresine ulaşıyoruz (Şekil 10-1’de gösterildiği gibi).



Gördüğünüz gibi, sayfa bize bir sürüm numarası (önceki ekran görüntüsünde X.Y.Z olarak gizledik), kütüphanenin nasıl kullanılacağı, kod örnekleri ve daha fazlasını sunuyor.

pubspec.yaml dosyasına eklemek:

Kullanım (Usage) bölümü, kütüphaneyi nasıl kullanmanız gerektiğini size anlatır. İndirme butonu veya kurulum işlemi yoktur. Aslında, neredeyse her Flutter/dart kütüphanesi için yapmanız gereken tek şey, pubspec.yaml dosyasını düzenleyip **dependencies** bölümüne bir satır eklemektir:

dependencies: flutter:

sdk: flutter

path\_provider: ^X.Y.Z

Önceki **caret (^)** simgesine dikkat edin. Bu, geliştirici araçlarına "Kütüphanenin en son sürümünü kullanabilirsin, ancak ana sürüm X'in üzerine çıkma" demek anlamına gelir. Bazı deneyimli Flutter geliştiricileri, geliştirici araçlarının hangi sürümü kullanacağına karar vermesinin riskli olduğunu düşündüklerinden caret simgesini kullanmazlar. Bu tamamen sizin tercihinize bağlıdır.

pubspec.yaml dosyanızı kaydeder kaydeder, geliştirici araçlarınız yeni kütüphaneyi algılayabilir ve sizin için indirebilir. Eğer yapmazsa, şu komutu çalıştırabilirsiniz:

$ flutter pub get

Running "flutter pub get" in myProject... 0.7s

$

Artık kütüphane indirildiğine göre, kullanmaya başlayabilirsiniz.

# X'in üzerinde çıkma ifadesi

Bir kütüphanenin sürüm numarasındaki **ana sürüm** (major version) X'in değerinin değişmemesi gerektiği. Flutter'da kütüphaneler sürüm numarasını genellikle şu şekilde belirtir: **X.Y.Z**. Burada:

* **X**: Ana sürüm numarası (major version)
* **Y**: Küçük sürüm numarası (minor version)
* **Z**: Yamanma sürümü numarası (patch version)

Örneğin, **path\_provider: ^1.2.3** yazdığınızda, Flutter araçları **1.x.x** sürümleriyle uyumlu olan kütüphaneleri kullanır, ancak **ana sürüm 2**'ye geçmez. Yani, **2.0.0** veya daha yüksek sürümlere çıkmaz. Bu, kütüphanenin büyük değişiklikler yapıp mevcut kodunuzla uyumsuz hale gelmesinin önüne geçer.

Bazı geliştiriciler **caret (^)** simgesini kullanmayarak daha sabit bir sürüm seçerler, böylece sadece belirli bir sürümdeki kütüphane kullanılır.

Kütüphaneyi içe aktarma:

Bu kütüphanenin işlevselliğine ihtiyaç duyduğun herhangi bir kaynak kodu dosyasını açın ve en üst kısmına bir içe aktarma (import) satırı ekleyin:

import 'package:flutter/material.dart'; import 'package:path\_provider/path\_provider.dart'; Kütüphaneyi kullanma:

Kullanımı, kütüphaneye ve tabii ki her kütüphanenin kendine özgü olmasına bağlıdır. Talimatlar ve örnekler yine dokümantasyonda bulunur.

path\_provider kütüphanesinin dokümantasyonu, **getTemporaryDirectory()** veya **getApplicationDocumentsDirectory()** adlı bir metodu çağırmamızı söyler, bu şekilde:

Directory documents = await getApplicationDocumentsDirectory();

File file = File('${documents.path}/$\_filename'); Futures, async ve await:

Flutter, Dart kullanılarak yazılmıştır ve Dart tek iş parçacıklı bir dildir. Bu, bir Flutter uygulamasının aynı anda sadece bir şey yapabileceği anlamına gelir, ancak bu, Flutter uygulamalarının daha yavaş işlemleri beklemek zorunda olduğu anlamına gelmez.

Flutter uygulamaları bir **etkinlik döngüsü** (event loop) kullanır. Bu, Android’in bir **main looper**'ı ve iOS’un bir **run loop**'u (yani ana döngü) olduğu düşünüldüğünde şaşırtıcı olmamalıdır. Hatta JavaScript geliştiricileri bile şaşırmaz çünkü JavaScript’in kendisi de… bekleyin… bir etkinlik döngüsüne sahiptir. Evet, şu anda bütün havalı geliştiriciler etkinlik döngüsü kullanıyor.

Bir etkinlik döngüsü, arka planda sürekli çalışan ve belirli aralıklarla uyanıp, çalışması gereken görevler için etkinlik kuyruğuna bakarak işlem yapan sonsuz bir döngüdür. Eğer herhangi bir görev varsa, etkinlik döngüsü bu görevleri çalıştırmak için yalnızca CPU boşta olduğunda çalıştırma yığınına ekler.

Uygulamanız talimatları çalıştırırken, bu talimatlar sırasıyla çalıştırılır – birbiri ardına. Eğer bir talimat, ana iş parçacığını bir kaynağı beklerken bloke edebilecekse, bu talimat başlatılır ve “bekleme” kısmı ayrı bir kuyruğa konur.

Neden beklesin ki?

Bazı şeyler, CPU'ya kıyasla yavaştır. Bir dosyadan okuma yavaştır. Bir dosyaya yazmak ise daha da yavaştır. Ajax ile iletişim kurmak mı? Bunu unutun.

Eğer bekleyen işlemi ana iş parçacığında tutarsak, bu tüm diğer komutları bloke ederdi. Ne kadar büyük bir israf olurdu!

JavaScript, iOS, Android ve şimdi de Dart'ta bu durum şu şekilde yönetilir:

1. Yavaş olması bilinen bir işlem başlatılır, normal şekilde çalıştırılır.
2. Bir şeyler beklemeye başladığı anda – disk, HTTP isteği, ne olursa – CPU'dan uzaklaştırılır.
3. Bir tür dinleyici (listener) oluşturulur. Bu dinleyici, işlemi izler ve bekleme tamamlandığında bir uyarı gönderir.
4. Bu dinleyicinin referansı, ana iş parçacığına geri döndürülür.

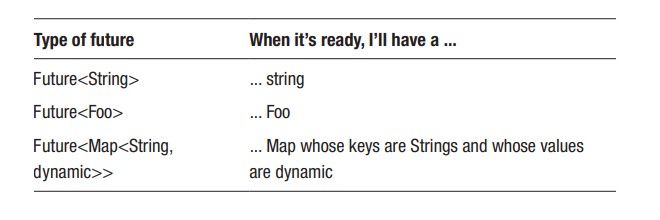
Bu referans nesnesine **Future** denir.

1. Ana iş parçacığı, kendi işine devam eder.
2. Bekleyen işlem nihayet çözüme kavuştuğunda, etkinlik döngüsü bunu görür ve ana iş parçacığında ilişkilendirilmiş bir metodu (yani bir geri çağırma fonksiyonunu) çalıştırarak yavaş işlemi tamamlar.

Yapmanız gereken tek şey, bir **Future** oluşturmak ve diğer metodlardan döndürülen **Future**'ları işlemek için kod yazmaktır.

// goReadAFile() fonksiyonu yavaştır ve bir Future döndürür Future myFuture = goReadAFile();

Dart'ta, bir **Future**'ın sonunda size vereceği şeyin türünü belirleme yeteneğiniz vardır:



O **Future** nesnesine sahip olduğumuzda, veriye sahip olmayabilirsiniz, ancak kesinlikle o veriyi gelecekte alacağınıza dair bir vaadinize sahip olursunuz. (Ne yaptıklarını gördünüz mü?) Bir **Future**'dan veriyi nasıl alırız?

Veri hazır olduğunda **Future**'a ne yapması gerektiğini söylersiniz. Temelde, “Yo, veri hazır” olayına yanıt veriyorsunuz ve **Future**'a ne yapması gerektiğini bir fonksiyon kaydederek söylüyorsunuz.

myFuture.then(myCallback);

**.then()** fonksiyonu, o geri çağırma fonksiyonunu kaydetme şeklinizdir. Geri çağırma fonksiyonu, vaat edilen veriyi işleme için yazılmalıdır. Örneğin, bir **Future<Foo>**'umuz varsa, geri çağırma fonksiyonumuzun imzası şöyle olmalıdır:

void myCallback(Foo theIncomingData) {

doSomethingWith(theIncomingData);

Yani eğer **Future** bir **Person** döndürecekse, geri çağırma fonksiyonunuz bir **Person** almalıdır. Eğer **Future** bir **String** vaat ediyorsa, geri çağırma fonksiyonunuz bir **String** almalıdır. Ve bu şekilde devam eder.

Geri çağırma fonksiyonlarınız her zaman **void** döndürmelidir çünkü **.then** fonksiyonu döndürülen bir değeri alamaz. Bunu düşündüğünüzde oldukça mantıklı olduğunu göreceksiniz, çünkü unutmayın ki artık uygulamanızın ana iş parçacığında çalışmıyor, dolayısıyla geri birleşmesi mümkün değildir. Peki, geri çağırmadan bir değer nasıl alırsınız? Birkaç yöntem vardır, ancak en anlaşılır olanı, geri çağırma fonksiyonunun dışında tanımlanan bir değişken kullanmaktır:

class FooState extends State<FooComponent> {

String \_firstName; // <-- Bütün sınıf tarafından bilinen bir değişken

Widget build(BuildContext context) {

// bir widget döndür

}

void \_myCallback(String someVar) {

\_firstName = someVar; // <-- Asenkron geri çağırmadan bir değer ALMA

}

}

**.then()** fonksiyonunu **Future** nesnesine eklemek zaman zaman kodunuzu karmaşıklaştırabilir. İsterseniz, biraz daha temiz hale getirmek için **await** kullanabilirsiniz.

Await:

Veriyi almak için daha doğrudan bir yol daha vardır, bu da okunması daha kolaydır. **.then()** kullanmak yerine, veriyi **await** edebilirsiniz.

Foo theIncomingData = await somethingThatReturnsAFuture();

**Awaiting** (beklemek), çalıştırılan kodu duraklatır ve **Future**'ın çözülmesini bekler, ardından bir sonraki satıra geçer. Önceki örnekte, **await** ettiğiniz "Foo" döndürülür ve **theIncomingData**'ya yerleştirilir.

async

İster sev ister sevme, bir fonksiyon içinde **await** kullandığınızda, o fonksiyon artık ana iş parçacığını bloke etme riski taşır, bu yüzden **async** olarak işaretlenmesi gerekir. Örneğin, bu fonksiyon:

Bar someFunction() {

Foo theIncomingData = someFunction();

return new Bar();

}

**Await** kullandığımızda şöyle olur:

Future<Bar> someFunction() async {

Foo theIncomingData = await somethingThatReturnsAFuture(); return new Bar();

}

İkinci satırda **await** eklediğimizde, fonksiyonu **async** olarak işaretlememiz gerektiğini unutmayın. İncelenen şey ise şu: **async** olarak işaretlendiğinde, o fonksiyondan dönen her şey, eğer zaten bir **Future** değilse, hemen bir **Future** içine sarılır.

Hazır mısınız? Şunu göz önünde bulundurun: Herhangi bir **Future**'ı **await** etmeye karar verdiğinizde, fonksiyon mutlaka **async** olarak işaretlenmeli, bu yüzden onu çağıran tüm fonksiyonlar da **await** edilmelidir ve **async** olarak işaretlenmelidir, ve bu böyle devam eder. Sonunda, çağrı zincirinde yeterince yüksek bir noktaya ulaştığınızda, artık bir fonksiyon içinde olmadığınız için **async** olarak işaretlemenize gerek kalmaz.

İpucu: **Flutter**'ın **build()** metodu **async** olamaz, ancak **onPress** gibi olaylar olabilir. Bu yüzden, bu geri dönümlü **async-awaitasync-await** durumunu çözmek için **async** işlemlerinizi olaylara yönlendirmeye çalışın.

İşte **Future**'larla ilgili önemli noktalar:

1. **Future**'lar, Dart kodunuzun **asenkron** olmasını sağlar – yavaş çalışan işlemleri ayrı bir iş parçacığında (bir tür) yönetebilir.
2. Bu işlemlerin geri çağırmalarını ya **.then(callback)** ile ya da **await** kullanarak işleyebilirsiniz.
3. Eğer bir fonksiyonda **await** kullanıyorsanız, o fonksiyon **async** olarak işaretlenmelidir.

**Future**'lar hakkında daha fazla okumak isterseniz, Dart ekibinin kapsamlı bir incelemesi burada: www.dartlang.org/tutorials/language/futures.