**11. BÖLÜM**

**RESTful API Çağrıları Yapmak için HTTP**

Artık bir Flutter uygulamasını nasıl oluşturacağımızı, harici kütüphaneleri nasıl kullanacağımızı ve JSON formatında verileri asenkron olarak nasıl okuyup yazacağımızı öğrendik. Bu aslında bizi sadece küçük bir adım daha ileri gidip bir API sunucusuyla veri alışverişinde bulunmamız için harika bir konuma getiriyor, gerçek dünya uygulamaları yazacaksak bu kesinlikle olmazsa olmazdır. Bilmemiz gerekenler şunlar:

• API çağrısı nedir?

• HTTP GET veya DELETE isteği yapmak.

• HTTP POST, PUT veya PATCH isteği yapmak.

• Yanıtı en basit şekilde işlemek.

• FutureBuilder ve StreamBuilder ile daha temiz işleme.

• Güçlü türde nesnelerle daha temiz işleme.

Bu bölüm için planımız bu olacak. Bu ve HTTP güncellemelerine izin veren bir web sitesiyle biraz pratik yapmak. Ve herkesin aynı sayfada olduğundan emin olmak için (kesinlikle kasıtlı bir kelime oyunu), muhtemelen bir API'nin tam olarak ne olduğuyla başlamalıyız. Zaten aşinaysanız, göz gezdirebilir veya tamamen atlayabilirsiniz.

**API çağrısı nedir?**

Uygulamanız zaten küçük, yerelleştirilmiş bir veritabanından okuma yeteneğine sahip.Ancak başka bir yerde bulunan bir veritabanından okuyamaz. Başka bir deyişle, Google veritabanına bağlanıp kayıtları okuyup yazamazsınız. Bu yapılamaz, veritabanı kimlik bilgileriniz olsa bile. Yani, herkes her yerden bağlanıp Google verilerini doğrudan değiştirebilseydi güvenlik açısından ne gibi sonuçlar doğuracağını düşünün. Yani, herkesin her yerden bağlanıp Google verilerini doğrudan değiştirebilmesi durumunu bir düşünün; güvenlik sonuçlarını bir düşünün. Bunun yerine geliştiriciler, verileri kontrollü bir şekilde okuyup yazabilmek için bir sunucu tarafı programı oluşturur ve bu programı, belirli bir protokol (genellikle https) ile belirli bir adreste İnternet üzerinden erişilebilir hale getirirler.

Bu sunucu tarafı verilerini okumak için, herhangi bir kullanıcı kullanıcı adı/şifre biçiminde kimlik bilgilerini veya daha iyisi, API anahtarı adı verilen benzersiz ve gizli bir anahtarı gönderdikten sonra HTTP istekleri yapabilir.

API terimi yine karşımıza çıktı. Uygulama Programlama Arayüzü anlamına geliyor. Farklı durumlarda farklı anlamlar ifade edebilir, ancak varsayılan anlamı, geliştiricilerin verileri okumak ve yazmak amacıyla HTTP istekleri gönderebileceği herhangi bir İnternet adresi olarak kabul edilmiştir. Pek çok kamuya açık API mevcuttur ve kendi API'nizi oluşturmak için birçok seçenek vardır. Bir API yanıt verdiğinde, genellikle JSON formatında olan bir veri akışı ile geri döner.

**API isteklerinin çeşitleri**

API sunucularıyla iletişim yalnızca birkaç farklı şekilde yapılır

(Tablo 11-1).

JSON hakkında daha fazla bilgi edinmek için burayı ziyaret edin: https://json.org

Tablo 11-1. HTTP yöntemleri ve açıklamaları

|  |  |  |
| --- | --- | --- |
| Http yöntemi | Amaç | Notlar |
| GET | Kayıtları okumak | Bir veritabanı okuma gibi, yalnızca bir sunucudan veri istemek |
| DELETE | kayıtları silmek | Sağlanan ID ile işaret edilen kaydı silme. Veri döndürülmez |
| POST | yeni kayıtlar oluşturmak | Zaten böyle bir kayıt olsa bile yeni bir kayıt oluşturma |
| PUT | mevcut kayıtların değiştirilmesi | Mevcut kaydı bu kayıtla değiştirme. Eski kaydı tamamen silip, yerine bu kaydı ekleme |
| PATCH | mevcut kayıtları güncelleme | Eski kaydı yerinde bırakma, ancak bu istekteki verilerle alanlarını güncelleme |

HEAD, CONNECT, OPTIONS, TRACE – Diğer istek türleri için. Tipik uygulamalar tarafından nadiren kullanılır. İsterseniz bunlar hakkında http://bit.ly/HTTPMethods Geliştiricilerin GET, POST, PUT, PATCH ve DELETE dışında bir şey kullanması nadirdir. Bunların hepsi Flutter'da http.dart'ı içe aktararak elde edeceğiniz bir Dart kütüphanesi kullanılarak yapılır.

İlk olarak, http paketini pubspec.yaml dosyanızın bağımlılıklar (dependencies) kısmına ekleyeceksiniz. "flutter pub get" komutunu ekleyip çalıştırdığınızda, paket <https://pub.dev/packages/http> adresinden indirilecektir.

import 'package:http/http.dart';

Bu, her HTTP yöntemine karşılık gelen yöntemleri olan http sınıfını açığa çıkaracaktır. Şimdi bu kütüphaneyi kullanarak istek göndermeye bakalım.

**HTTP GET veya DELETE isteği yapma**

İlk önce GET ve DELETE istekleriyle başlayacağız çünkü bunlar en basit olanlardır; hiçbir zaman gövdeleri olmaz.2 Aslında, tek karmaşıklık HTTP isteklerinin eşzamansız olarak yapılmasıdır. Bunlar bir Future döndürür ve bunu ya .then() ile işlemeniz ya da beklemeniz gerekir. Bu yüzden isteğinizi şu şekilde yapabilirsiniz:

String url = 'https://us.com/people/1234';

Response response = await get(url);

print(response.statusCode); // 200, we hope

Map<String, dynamic> person = json.decode(response.body);

print(person['first']);

print(person['last']);

print(person['imageUrl']);

print(person['email']);

or with a .then() like this:

get(url).then((Response res) {

print(res.statusCode); // 200, we hope

Map<String, dynamic> person = jsonDecode(res.body);

print(person['first']);

print(person['last']);

print(person['imageUrl']);

print(person['email']);

});

Silme istekleri aynı şekilde yapılır. Aslında, genellikle yanıt değeri olmadığı için daha basit olabilirler. DELETE isteği, genellikle bir kaydı silmek için kullanılır ve başarılı bir silme işlemi genellikle herhangi bir veri döndürmez.

2 Bu, oldukça tartışmalı bir konudur. HTTP spesifikasyonu bu konuda sessizken, RFC 2616, DELETE isteğinde bir gövdenin göz ardı edileceğini ima eder, ancak açıkça yasaklamaz. Bazı sunucular gövdeyi göz ardı eder. Diğerleri ise tüm isteği göz ardı eder. Bazıları ise 400 hata kodu döndürebilir. Sunucunuzda test edebilir ya da güvenli tarafta kalıp gövdeyi atlayabilirsiniz.

Ya başarılı olur ve geri değer döndürmez ya da 400 veya 500 serisi bir yanıtla başarısız olur.

Response response = await delete(url);

Dikkat: Herhangi bir HTTP isteği yaparken, URL'yi göndermeden önce her zaman kodlamalısınız. Bu, URL'nin geçerli olmasını sağlamaya yardımcı olacak ve özellikle kullanıcıdan giriş alırken güvenliği artırabilir. Uri.encodeFull'ı şu şekilde çağırabilirsiniz:

String url = Uri.encodeFull('http://us.com/api/ppl?query=Jo Ki');

Basitlik açısından, örneklerde kodlamayı atlayacağız. Ancak gerçek dünyada, her zaman bunu yapmayı unutmayın.

**HTTP PUT, POST veya PATCH isteği oluşturma**

PUT, POST ve PATCH, GET ve DELETE ile oldukça benzerdir. En büyük fark, PUT, POST ve PATCH'in her zaman bir gövde (body) gerektirmesidir – genellikle JSON formatında anahtarlar ve değerler içeren bir dizedir.

String payload = '{"first":"Kamala", "last":"Khan", "id":374}'; Response response = await post(url, body:payload);

Bu yanıt, GET ve DELETE isteklerinde olduğu gibi açılır.

**NOT** POST, PUT ve PATCH ile, veriyi istemciden sunucuya gönderiyoruz. Sunucuya hangi şekilde veri gönderdiğimizi belirtmek uygun ve bazen gereklidir. Bunu, isteğe bir HTTP başlığı ekleyerek yapacağız. "ContentType" adlı bir anahtar sağlayacağız ve değerini "application/json" olarak belirleyeceğiz. Bunu şu şekilde yaparız: Map headers= {'Content

Type':'application/json'};

Response res = await post(url, headers:headers,

body:payload);

Bu konuyu işlerken, başlıklar içinde yararlı olabilecek birçok değişken olduğunu unutmayın; örneğin Accept, Accept-Encoding, Authorization, Content-MD5, Cookie, Date, Host, If-Modified-Since ve diğerleri. Bunlar hakkında daha fazla bilgiye burada ulaşabilirsiniz:

<https://en.wikipedia.org/wiki/> List\_of\_HTTP\_header\_fields#Request\_fields.

API'den HTTP istekleri yapmak o kadar da zor değildi, değil mi? Çok hızlı bir şekilde, Flutter uygulamalarımızı istek yapabilen, yanıtı deserialize edebilen ve bunu debug konsoluna yazdırabilen hale getirdik. Ama Flutter, bu veriyi havalı görünümlü widget'lar içinde görüntülemekle ilgili. Peki, bu istekleri widget'lara nasıl entegre ederiz?

**HTTP Yanıtlarını Widget'lara Entegre Etmek**

Geleceğin çözülmesini bekleyip sonra onu göstermenin bir avuç yolu var. Size sadece üçünü göstererek işleri basitleştireceğiz, kaba kuvvet yolu, FutureBuilder ve StreamBuilder. Kaba kuvvet açıktır ve anlaşılması kolaydır, ancak FutureBuilder/StreamBuilder'ı daha temiz ve daha zarif oldukları için seveceğinizi düşünüyorum.

**Kolay yöntem – Kolay yol (Brute force – The easy way)**

Verileri görüntülemek için ihtiyaç duyduğunuz tüm araçlara zaten sahipsiniz: Futures'ı anlıyorsunuz ve durumlu widget'a yeni verilerle kendini yeniden çizmesini nasıl söyleyeceğinizi biliyorsunuz – setState(). Yani bu, .then()'in içine veya await'in sonrasına bir setState() koymak kadar basit olabilir:

String url = 'http://us.com/api/people/12345';

Response response = await get(url);

Map<String, dynamic> responseBody = json.decode(response.body);

String first = responseBody['first'];

String last = responseBody['last'];

String imageUrl = responseBody['profilePictureUrl'];

Widget card = Stack(

children: <Widget>[

Image.network(imageUrl,

height: 300, width: 300, fit: BoxFit.cover),

Text("$first $last"),

],

);

setState(() {

\_cardWidget = card;

});

Ve tabii ki yapı yönteminiz \_cardWidget'ı bir yerde görüntülediği sürece, Future çözülür çözülmez uygun verilerle işlenecektir; bu da yalnızca HTTP GET isteği veri döndürdüğünde gerçekleşir. Çocuk oyuncağı! Ama en zarif şey değil.

**Temiz yöntem- Temiz yol (FutureBuilder – The clean way)**

Daha iyi bir çözüm FutureBuilder widget'ı olabilir. Eğer bir Future'ınız varsa ve bu Future'ın yerine getirildiğinde Flutter widget'ında işlenmesi gereken verilere sahip olduğu bir noktadaysanız FutureBuilder'a bakın. Bu senaryo tanıdık geliyor mu? Gelmelidir çünkü Flutter'da Future'lara sahip olmamızın başlıca nedeni budur. Daha önceki basit kod örneği, FutureBuilder ile bunun gibi çok daha eksiksiz bir şekilde yapılabilir:

FutureBuilder(

future: get(url),

builder: (BuildContext ctx, AsyncSnapshot<dynamic> snapshot) {

if (snapshot.connectionState != ConnectionState.done) {

return const CircularProgressIndicator();

}

if (snapshot.hasError) {

return Text('Oh no! Error! ${snapshot.error}');

}

if (!snapshot.hasData) {

return const Text('Nothing to show');

}

final Map<String, dynamic> responseBody =

json.decode(snapshot.data.body);

final int statusCode = snapshot.data.statusCode;

if (statusCode > 299) {

return Text('Server error: $statusCode');

}

String first = responseBody['first'];

String last = responseBody['last'];

String imageUrl = responseBody['profilePictureUrl'];

return Stack(

children: <Widget>[

Image.network(imageUrl,

height: 300, width: 300, fit: BoxFit.cover),

Text("$first $last"),

],

);

},

);

setState() kullanmaya gerek yoktur çünkü FutureBuilder kendisi doğrudan Future'a erişebilir, böylece ne zaman ve nasıl yeniden çizileceğini bilir. Önceki örnekte, her durum için farklı bir şey nasıl render edebildiğini görebilirsiniz: Future'ın çözülmesini beklerken bir ProgressIndicator (yükleniyor göstergesi), bir hata durumu varsa hata mesajı, Future içinde hiçbir şey yoksa bir bildirim ve tabii ki veri başarılı bir şekilde geldiğinde widget.

**Dikkat** Her zaman snapshot.hasData ve/veya snapshot.has Error'ı kontrol edin anlık görüntü.data'ya erişmeden önce. Yazma zamanında, flutter bir hata varsa fırlatmaz. Bunun yerine, hatayı yutar ve sessizce başarısız olur.

Ayrıca response.statusCode'da bulunabilen durum koduna dikkat edin! Bu sayı 400'lerde veya 500'lerdeyse, sunucudan geçerli bir yanıt almışsınızdır, ancak bu bir sorundur ve verileriniz boş olacaktır.

**Akış Oluşturucu (StreamBuilder)**

FutureBuilder'ın geleceklerle yaptığını StreamBuilder akışlarla yapar. Bu iki sınıf neredeyse aynıdır, aynı formata sahiptir, aynı anlık görüntü şeklini kullanır ve snapshot.hasErrors ve snapshot.hasData'yı kontrol eder. Ancak bazen bir gelecekle olduğu gibi tek bir veri dönüşüyle ​​değil, bize ani veya dalgalar halinde gelebilecek bir veri akışıyla uğraşırız. Durum böyle olduğunda, bunun yerine bir StreamBuilder kullanmak isteyeceksiniz:

StreamBuilder(

stream: anythingThatReturnsAStream(),

builder: (BuildContext ctx, AsyncSnapshot<dynamic> snapshot) {

// Everything below this is pretty much the

// same as FutureBuilder but the data is a

// collection of documents, each being a record

if (snapshot.connectionState != ConnectionState.done) {

return const CircularProgressIndicator();

}

if (snapshot.hasError) {

return Text('Oh no! Error! ${snapshot.error}');

}

if (!snapshot.hasData) {

return const Text('Nothing yet. Please wait ...');

}

return ListView.builder(

itemCount: snapshot.data.documents.length,

itemBuilder: (BuildContext context, int i) {

String first = snapshot.data.documents[i]['first'];

String last = snapshot.data.documents[i]['last'];

String imageUrl = snapshot.data.documents[i]

['imageUrl'];

return Stack(

children: <Widget>[

Image.network(imageUrl,

height: 300, width: 300, fit: BoxFit.cover),

Text("$first $last"),

],

),

},

);

);

},

);

Bu tür kodlar yazmayı unutmayın, yeni gelen verilere göre uyanan ve kendini güncelleyen kodun bir terimi vardır: reaktif programlama. Etkin programlama, uygulamamızı dış etkilerden haberdar ettiğimizde ve ona bir şekilde tepki vermesini söylediğimizde gerçekleşir. RxJava, rxJS ve rx gibi reaktif uzantıları duymuş olabilirsiniz. Bu stil için yapılmış sınıflar ve işlevlere sahip kütüphanelerdir. Şaşırtıcı olmayan bir şekilde lutter için rxDart adında bir tane var. Bunu şu adreste bulabilirsiniz: <https://github.com/ReactiveX/rxdart>.

**Güçlü tipli sınıflar** (**Strongly typed classes)**

Şu anda bir API'ye karşı HTTP istekleri yapmayı biliyorsunuz ve bir yanıt aldığınızda, o veriyi nasıl çözüp kullanacağınızı da öğrenmiş oldunuz. Bu, veriyi güçlü tipli bir sınıfa dönüştürmek için tipli deserialization (veri çözümleme) desenini kullanmaya başlamak için harika bir durumdayız.

**NOT** Bunun çağrı yapmak için gerekli olmadığını unutmayın. Bu yalnızca çağrıyı işlemenin ve onu öngörülebilir bir yapıya çekmenin daha temiz bir yoludur. Veriler doğası gereği yapılandırılmamıştır. Bu, birçok lutter geliştiricisi tarafından kullanılan en iyi uygulamadır ancak kesinlikle gerekli değildir. Bu nedenle, beğenmezseniz, atlayabilirsiniz.

Yazılı deserializasyon üç basit adımda gerçekleşir:

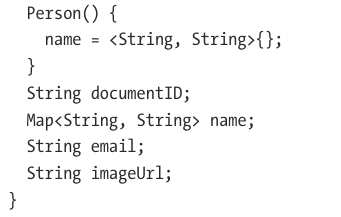
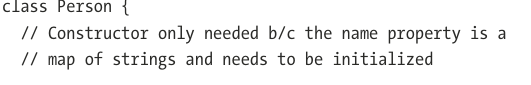
1. İş sınıfını oluşturun.

2. Bir .fromJSON() yöntemi ve/veya bir .fromJSONArray() yöntemi yazın.

3. HTTP çağrılarından okurken, nesneyi nemlendirmek için .fromJSON() kullanın.

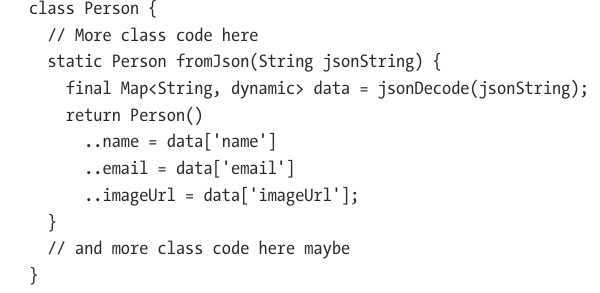
**Bir işletme sınıfı yaratın**

Diyelim ki insanlar için veri okuyor ve yazıyoruz. Bir PERSON sınıfı oluşturmalıyız:



**Bir .fromJSON( ) yöntemi yazın**

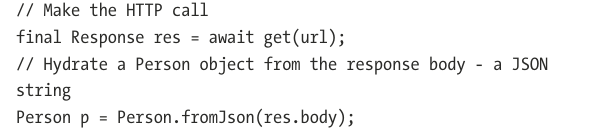
Bu, iş sınıfının bir örneğini döndürecek statik bir yöntem olmalıdır, bu durumda Person:



Dart'ın basamaklı operatörlerinin kullanımına ve yeni operatörünün atlanmasına dikkat edin. Her ikisi de en iyi uygulamadır.

**Nesneyi hidratlamak(**hydrate**) için .fromJSON( ) kullanın**

Hidrat kelimesi tam anlamıyla "su eklemek" anlamına gelir. Bu bağlamda, veri sudur ve biz de veriyi ona ekleyerek yeni bir Kişi nesnesi yaratıyoruz. .get() metodunu kullanarak bir HTTP servisinden veri okursunuz ve bunu .fromJSON()'a şu şekilde geçirirsiniz:



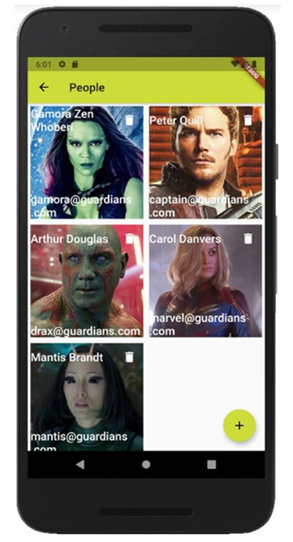
Kodun ne kadar temiz ve anlaşılır olduğunu görüyor musunuz?

Sanırım bu noktada, tüm bu yeni edinilen bilgileri kullanmak istersiniz. Hadi bunu bir sonraki adımda ücretsiz bir API hizmetiyle yapalım.

**Büyük bir örnek**

Gerçek bir API hizmeti, sunucuda bazı kapsamlı kurulumlar gerektiren GET, POST,DELETE, PUT ve/veya PATCH uç noktalarına sahip bir veritabanını içerecektir. Sonunda oraya varmak isteyeceksiniz ve size bir sonraki bölümde harika bir kalıcı çözüm göstereceğiz. Ancak şimdilik, hiçbir maliyeti olmayan ve kurulumu kolay, yalnızca gösterim amaçlı bir siteden yararlanalım, böylece API verilerini kullanan bir Flutter uygulaması oluşturabiliriz.

Bir insan bakım CRUD uygulaması oluşturalım (Şekil11-1).



Şekil 11-1. RESTful API'den verilerini okuyan ve yazan kayıtları tutan bir uygulama

Bu uygulama, REST prensiplerine uyan bir HTTP web servisinden bir kişi listesini okuyacaktır; bu, diğer şeylerin yanı sıra, daha önce öğrendiğimiz HTTP GET, POST, PUT ve DELETE yöntemlerini desteklediği anlamına gelir. Ek olarak, tüm kayıtlar, bu kayıtların oluşturulmasının bir parçası olarak sunucuda kendilerine atanan benzersiz bir kimliğe sahip olacak. Kişi listemizi göstermek için önceki resimli sahneyi oluşturacağız. Ayrıca kişinin adını, e-posta adresini ve fotoğrafın çekildiği yeri girebileceğimiz alanlara sahip başka bir sahneye de ihtiyacımız olacak.

Kulağa eğlenceli geliyor mu? Sunucuyu kurduktan sonra eğlenceli olacak. Hadi bununla başlayalım.

**Kurulum**

Açıkça bir sunucuya ihtiyacımız var. Bu yüzden bir tane kurmamız ve 80 numaralı portta HTTP istekleri alan bir veritabanı ve bir servis kurmamız yapılandırmamız ve bir sürü başka şey gerekecek.

Pipedream'i tanıtıyoruz!

Pipedream.com, deneyimsiz geliştiricilerin bile kolayca karmaşık süreçler oluşturabileceği bulut tabanlı iş akışları sağlar. Mükemmel hizmetleri, makul miktarda veri ve işlem için de ücretsizdir. Pipedream'deki iyi insanlar, çok az çabayla ve hiçbir maliyet olmadan, halka açık bir RESTful API üzerinden veri okumamızı ve yazmamızı sağlayan basit bir RESTful arayüzü ortaya çıkaran bir iş akışı oluşturdular.

Tüm bunları anonim olarak yapabilirsiniz ancak daha sonra verileri tekrar inceleyebilmeniz için bir hesaba giriş yapmanızı öneririz. Tarayıcınızda <http://pipedream.com> adresini açın. Bir şifreyle bir hesap oluşturun.

Bu bağlantıyı ziyaret edin: <http://bit.ly/pipedream_api> Sağ üstteki büyük "Fork" düğmesine basın. Bu, ihtiyacınız olduğunda sahip olabileceğiniz ve değiştirebileceğiniz kendi kopyanızı oluşturacaktır. Sayfanın en üstünde bir URL göreceksiniz. Bu, istekleri göndereceğiniz URL'dir.



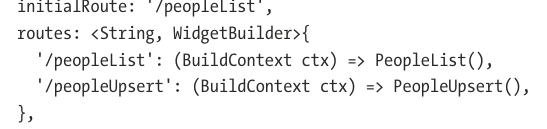
Bunu test etmek için bir komut satırına gidin. Kurulumunuzun tamamlandığından emin olmak için curl komutunu kullanın.



Sunucudan “Başarılı” yanıtını alırsanız, işlem çalışıyor demektir ve devam edebilirsiniz.

**Flutter uygulamasını oluşturun**

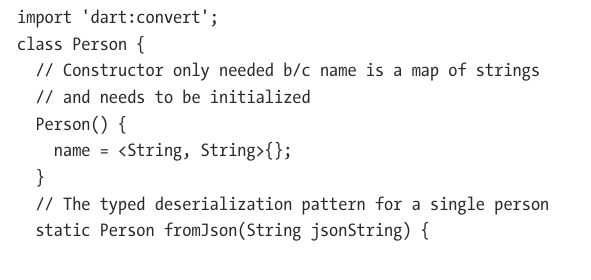
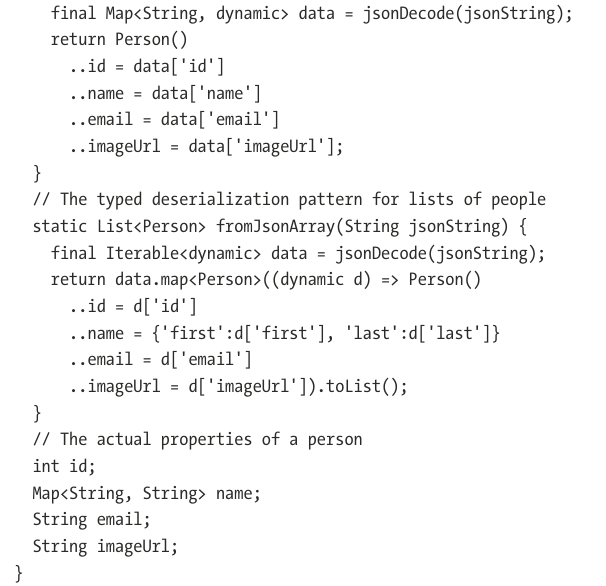
Flutter create kullanarak yeni bir Flutter uygulaması oluşturun. main.dart'ı açın ve MaterialApp widget'ınızı bulun. "Home" özelliğini kaldırın ve buna şunu ekleyin:



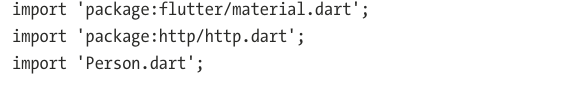
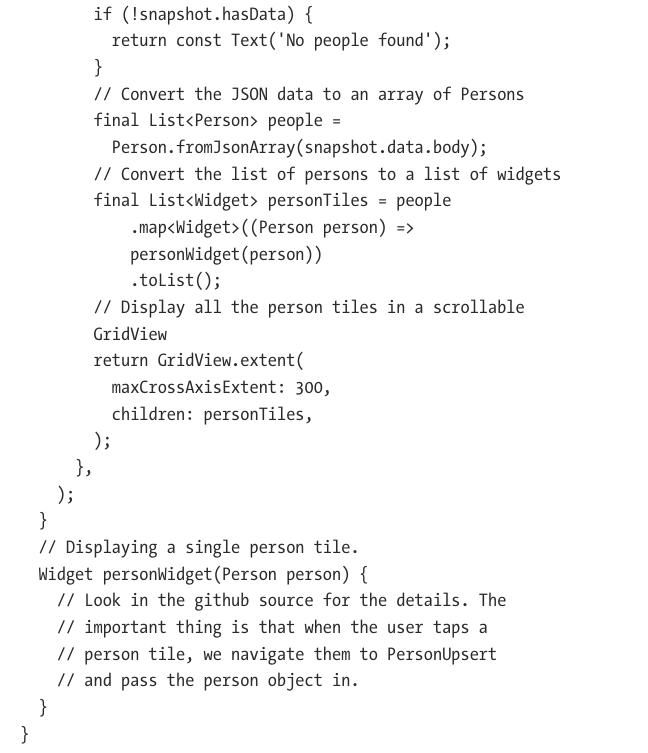
Daha sonra iki yeni StatelessWidget oluşturun, biri “PeopleList.dart”, diğeri ise “PeopleUpsert.dart” adını taşısın. Ayrıntılarını bir dakika içinde dolduracağız. Ancak önce, bir Kişi nesnesini temsil eden bir iş sınıfı oluşturmak iyi bir fikir olabilir.

**Güçlü bir şekilde yazılmış bir işletme sınıfı oluşturma**

Kişilerle çalıştığımız için, her kişiyi tutmak için bir Kişi sınıfı oluşturmak iyi bir fikir olabilir. Bu çok isteğe bağlı en iyi uygulama, sunucu verilerini serileştirme ve serileştirmeyi kaldırma hatalarından kaçınmamıza yardımcı olabilir ve tüm Kişiyle ilgili mantığımızı yönetmek için merkezi bir yer sağlayabilir:

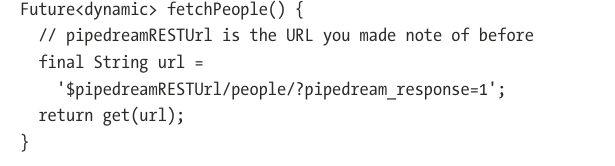
  


**PeopleList.dart**

Sonunda RESTful hizmetinden bir kişi listesi okuyacağız ve verilerini görüntülemek isteyeceğiz. PeopleList widget'ı, bu kişi listesini göstermekten sorumludur:  


**Flutter'da bir GET isteği**

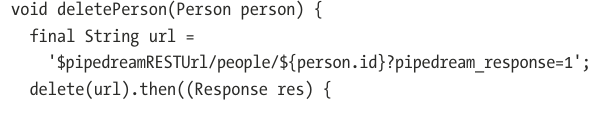
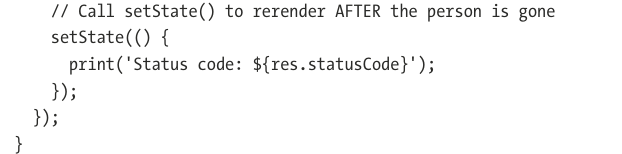
getScaffoldBody() metoduna geri bakın. Bir FutureBuilder'ı vardır. Future özelliği, yalnızca Person kayıtlarının bir JSON dizisiyle yanıt verecek URL'ye bir GET isteği yapması gereken fetchPeople() adlı bir metodu işaret eder:



Flutter altyapısını oluşturduktan sonra GET isteği oldukça basit, değil mi? Elbette bu çağrıyı ilk kez yaptığınızda, henüz herhangi bir kişi oluşturmadığınız için hiçbir şey görünmeyecektir. Bu yakında yeterince görünecektir.

**Flutter'da bir DELETE isteği**

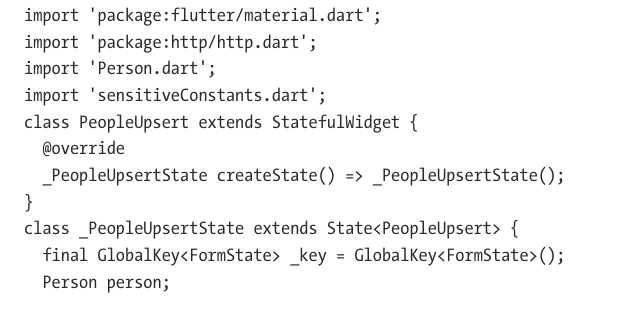
Her kişi döşemesinin sağ üst köşesinde bir çöp kutusu IconButton'ı vardır. Üzerine bir dokunuş deletePerson()'u çağırır ve kurtulmak istediğimiz kişiyi alır. Bu deletePerson() yöntemi, o kişiyi ID'sine göre işaret eden bir HTTP DELETE isteği göndermelidir:

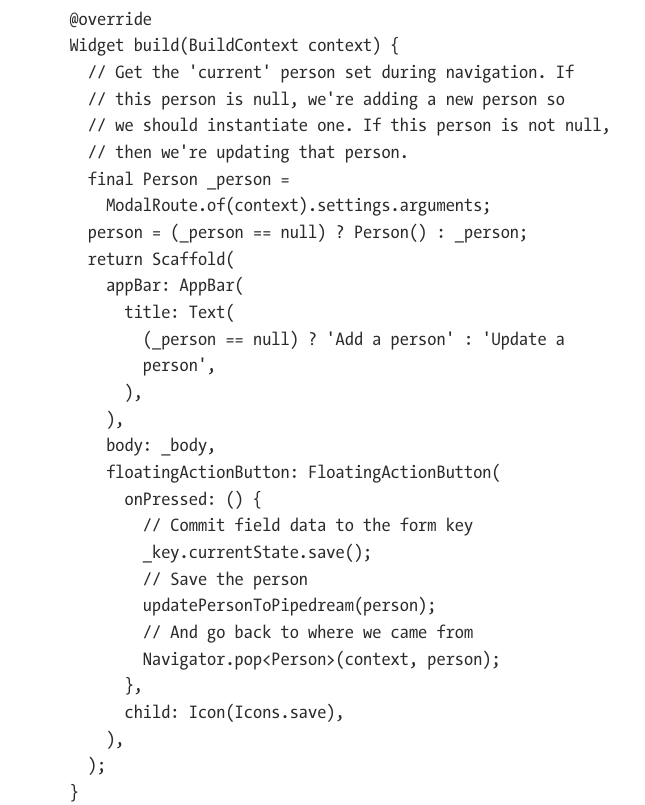
Silinenlerin Future'ı çözüldükten sonra, sahnenin yeniden oluşturulmasını zorlamak için setState()'i çağırdığımızı ve böylece Kişiler listesini yenilediğimizi unutmayın.

**PeopleUpsert.dart**

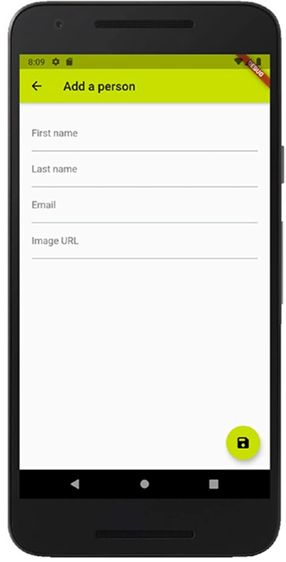
PeopleList'te kişileri okuma ve bir kişiyi silme işini hallettik. Ancak yeni bir kişi eklemek için kullanıcının bilgi girmesi için bir form gerekecek. Dikkatli okuyucular, mevcut kişileri güncellemek için aynı formun gerekli olduğunu fark edecektir. DRY ilkesine⁵ uymak için bir form oluşturalım ve hem ekleme hem de güncelleme için yeniden kullanalım:



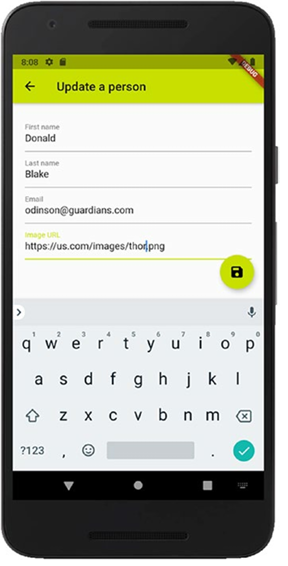
⁵ <https://en.wikipedia.org/wiki/Don%27t_repeat_yourself>


Bu, Şekil 11-2 veya 11-3'teki resimlerden birini oluşturacaktır.

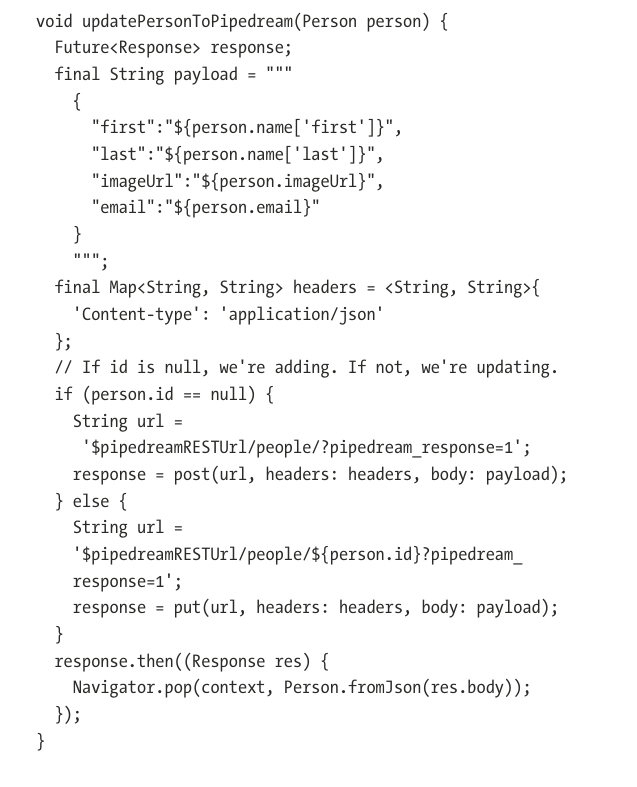


Şekil 11-2. Kullanıcı "+" düğmesine dokunmuşsa, şunu ekliyoruz:



Şekil 11-3. Kullanıcı bir kişi döşemesine dokunduysa, o kişiyi güncelliyoruz.

Flutter'da bir POST ve PUT isteği Kullanıcı PeopleUpsert sahnesinde FAB'a bastığında, girdiği verileri taahhüt eder ve updatePersonToPipedream()'i çağırırız. Eğer bir Add işlemiyse, bir POST çağrısı yapmak isteriz. Eğer bir Update işlemiyse, bir PUT çağrısı yapmak isteriz:



**Sonuçlar**

Hiç fena değil, değil mi? HTTP üzerinden veri okuma ve yazma hakkında neredeyse hiçbir şey bilmemekten, yazılmış serileştirme deseni ve FutureBuilder widget'ı gibi oldukça gelişmiş teknikler kullanan kapsamlı bir örneğe geçtik.

Ancak ne yazık ki, örneklerimiz üretim uygulaması için uygun olmayan geçici bir sunucu tarafı çözümü kullanıyor. iOS, Android ve Web'de Flutter ile sorunsuz çalışan sağlam, kalıcı bir sunucu tarafı veritabanıyla ilgilenir misiniz? Evet mi? O zaman sayfayı çevirin ve Google Firebase hakkında bilgi edinelim