**BÖLÜM9**

# DURUM YÖNETİMİ

Bu konuyu ilk bölümden beri ima ediyorduk çünkü sürekli **StatelessWidget** genişleten sınıflar yazıyorduk. Şimdi, Flutter'da bir **StatelessWidget** varsa, muhtemelen bir **StatefulWidget** de vardır diye düşünüyorsunuzdur. Ve haklısınız.Peki, **StatefulWidget** tam olarak nedir? **StatelessWidget** ile arasındaki fark nedir? Hangisini, ne zaman tercih etmeliyiz? **StatefulWidget**'in yapısı nasıldır? Kullanımıyla ilgili kurallar var mı? Veriler değişirse, bileşeni nasıl yeniden çizeriz?   
İyi sorular, değil mi? Sabırlı olun genç Jedi, bu bölümde bunların hepsini ve daha fazlasını cevaplayacağız.

### Durum (State) Nedir?

*Durum, değiştiğinde widget'ın yeniden çizilmesini gerektiren veridir.* —Rap Payne ;-)

StatelessWidget'ler veri içerebilir, ancak bu veri ya değişmez ya da widget aktifken ekranın görünümünü değiştirmez. Elbette, Flutter widget'ı yok edip yeniden oluşturduğunda veri değişebilir, ancak bu durum sayılmaz. Bir verinin **durum (state)** olarak kabul edilmesi için widget aktifken değişmesi ve güncel kalabilmesi için yeniden çizimi gerektirmesi gerekir.

© Rap Payne 2019

R. Payne, *Beginning App Development with Flutter*, https://doi.org/10.1007/978-1-4842-5181-2\_9

187

Flutter, bize kutudan çıkar çıkmaz **stateful** olan bazı widget'lar sunar.

* Uygulama Çubuğu
* [Alt](https://docs.flutter.io/flutter/material/BottomNavigationBar-class.html) navigasyon çubuğu
* OnayKutusu
* [Varsayılan](https://docs.flutter.io/flutter/material/DefaultTabController-class.html) sekme denetleyicisi
* Kapatılabilir
* [Menü](https://docs.flutter.io/flutter/material/DrawerController-class.html) Denetleyicisi
* [Açılır](https://docs.flutter.io/flutter/material/DropdownButton-class.html) Menü Düğmesi
* [Düzenlenebilir](https://docs.flutter.io/flutter/widgets/EditableText-class.html) Metin
* [Form](https://docs.flutter.io/flutter/widgets/Form-class.html)
* Form Alanı
* [Parlayan](https://docs.flutter.io/flutter/widgets/GlowingOverscrollIndicator-class.html) Aşım Göstergesi
* [Görsel](https://docs.flutter.io/flutter/widgets/Image-class.html)
* [Girdi](https://docs.flutter.io/flutter/material/InputDecorator-class.html) Süsleyici
* Ay seçici
* Gezgin
* İlerleme Göstergesi
* Radyo
* Yenileme Göstergesi
* [Çerçeve](https://docs.flutter.io/flutter/material/Scrollbar-class.html)
* [Kaydırıcı](https://docs.flutter.io/flutter/material/Slider-class.html)
* [Anahtar](https://docs.flutter.io/flutter/material/Switch-class.html)
* [Metin](https://docs.flutter.io/flutter/material/TextField-class.html) Alanı
* [Yıl](https://docs.flutter.io/flutter/material/YearPicker-class.html) seçici

... ve daha birçokları. Bunların hepsi, verilerin değişmesiyle birlikte widget'ı yeniden çizmek için izlenmesi ve korunması gereken dahili verilere sahiptir.  
Basit bir örnek alalım: bir **TextField** widget'ı. Evet, Web'deki bir metin kutusuna benzeyen, kullanıcıların içine karakter yazabileceği yerleşik widget'tan bahsediyoruz. Elbette, kullanıcı yazdıkça widget'ın ne yazdıklarını takip ettiğini ve görüntülediğini fark ediyorsunuz. İşte, dostum, bu **durum** (state).  
Bu harika, ama kendi **StatefulWidget**'larımızı nasıl yazacağız?  
Devam edin!

**StatefulWidget'ta neler bulunur?**

İşte bir StatefulWidget'ın şekli:

class Foo extends StatefulWidget { @override

\_FooState createState() => \_FooState();

}

class \_FooState extends State<Foo> {

//Private variables here are considered the 'state' @override

Widget build(BuildContext context) { return someWidget;

}

}

Bir **stateful** widget oldukça karmaşık görünebilir, ancak yapısına alıştıkça ikinci doğa haline gelir. Genellikle tek bir Dart dosyasında yazılır, ancak her zaman iki sınıftan oluşur: widget sınıfı ve state sınıfı.  
Widget sınıfı, **StatefulWidget**'ten türetilir ve başkaları tarafından kullanılacağı için herkese açıktır.  
State sınıfı ise her zaman özel (private) olur çünkü şu anki widget dışında bu sınıfı görecek başka bir şey yoktur. **State** sınıfı, şunlardan sorumludur:

1. **Durum verilerini tanımlamak ve yönetmek.**
2. **build()** metodunu tanımlamak – Widget'ı ekranda nasıl çizeceğini bilir.
3. Veri toplama veya olay işleme için gerekli olan geri çağırma (callback) fonksiyonlarını tanımlamak.

Peki, widget sınıfı ne yapar? Pek bir şey değil. Widget sınıfı sadece biraz kenara çekilir.

So then why separate them? There are two reasons. First, the [single](https://en.wikipedia.org/wiki/Single_responsibility_principle)  [responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)1 (the SRP) suggests that we should have one thing responsible for drawing the widget and another thing responsible for dealing with data. That’s just good software design. Other frameworks suggest that you separate UI from state management, but most don’t enforce it. Flutter does.

Second is performance. Redrawing takes time. Recalculating state takes time. When we separate them like this, we are giving the processor a chance to handle these two things independently. Sometimes a redraw doesn’t need to happen just because state changes. So we save the cycles of redrawing.

Also, when we redraw, Flutter creates and draws a whole new widget. The old widget in memory is no longer needed so it is dereferenced and eventually garbage collected. That’s awesome but state is still needed.

If Flutter retains that old state object, it can be reused instead of being garbage collected and recreated. By separating these objects, Flutter decouples them so they can each be handled in its own most efficient way. It’s a brilliant design!

### The most important rule about state!

When you change any state value, you should do it ...

1. In the state class
2. Inside a function call to setState():

setState(() {

// Make all changes to state variables here...

\_value = 42; // <-- ... Like this

});

1<https://en.wikipedia.org/wiki/Single_responsibility_principle>

**setState()**, bir fonksiyon alır ve bu fonksiyon ... eh ... kısa bir süre içinde çalıştırılır. Flutter alt sistemi, değişiklikleri gruplar ve bunları en uygun olduğunu düşündüğü bir zamanda çalıştırır. Bu, son derece verimlidir çünkü, diğer nedenlerin yanı sıra, ekranın yeniden çizilme sayısını azaltır.  
 **setState()**, değişkenleri en verimli ve kontrollü şekilde ayarlamakla kalmaz, aynı zamanda bu widget'ın yeniden çizilmesini zorunlu kılar. **build()** metodunu sahne arkasında çağırır. Sonuç olarak: Bir değeri değiştirdiğinizde, widget kendisini yeniden çizer ve kullanıcı yeni sürümü görür. Bu widget içinde alt widget'lar (yani iç widget'lar) varsa, bunlar **build()** metodunda bulunur, bu nedenle **setState()** çağrısı aslında bu widget'ın içindeki her şeyi, tüm alt ağaçları dahil olmak üzere yeniden çizer.  
 Eğer bu sizi bir an için paniğe sokarsa, lütfen unutmayın ki Flutter, sanal bir widget ağacı kullanır, bu yüzden her şeyi çizmesi gerektiğini söylesek de, ekranın hangi kısımlarının yenilenmesi gerekmediğini bilmeye yetecek kadar akıllıdır ve sadece gerçekten yenilenmesi gereken bölümleri teknik olarak yeniden çizer. Bu, süper verimlidir!

**Geçiş durumu**

Tamam, beni yakaladınız. Teknik olarak, bir ana widget'tan iç widget'a durum (state) geçiremezsiniz çünkü durum yalnızca bir widget içinde var olur. Ancak, veriyi aşağıya doğru geçirmemiz kesinlikle gerekir. Bu veri, ana widget'taki **stateful** veri olabilir ve iç widget'taki **state**'e taşınabilir.

Ama bu yeni bir şey değil. Bunu **stateless** widget'larla da yapmıştık. Hatırlatmak gerekirse, sınıf kapsamındaki final değişkenleri basitçe tanımlarsınız ve başlangıç değerlerini constructor parametrelerinde sağlarsınız.

Peki, geçirilen değer **State** sınıfında nasıl görünür? Flutter, StatefulWidget'ı temsil eden bir **widget** adında bir nesne sunar. Başka bir deyişle, eğer StatefulWidget'ta "x" adında bir değişken varsa, **State** sınıfında bu değişken "widget.x" olarak görünür.

class Foo extends StatefulWidget { final String passedIn;

// Value passed in from its host

ColorValueChanger({Key key, this.passedIn}) : super(key: key);

\_FooState createState() => new \_FooState();

}

class \_FooState extends State<Foo> { @override

Widget build(BuildContext context) { return Text(widget.passedIn,);

}

}

Şimdi, veri iletimini ana widget'tan iç widget'a nasıl yapacağımızı öğrendik, hadi diğer yöne bakalım ve veriyi iç widget'tan ana widget'a nasıl geri iletebileceğimizi görelim.

**Durum yedeklemesi kaldırma**

Ve yine beni yakaladınız. Durumu geçiremezsiniz. Ama işler daha da kötüleşiyor. Flutter ile, hiçbir şeyi yukarıya geçiremezsiniz.  
 Flutter, tek yönlü veri akışına sahiptir. Nokta. Veri yalnızca bir ana widget'tan iç widget'a akabilir. Bunu yaklaşık 200 sayfa boyunca yapıyoruz, değil mi? Ama bazen verinin iç widget'tan ana widget'a geri akması gerekir.  
Örneğin, bir **Login.dart** widget'ımız olduğunu varsayalım, içinde kullanıcı adı/şifre metin alanları ve bir gönder butonu bulunuyor. Bu **Login** widget'ını, kullanıcı zaten giriş yapmamışsa diğer widget'larda kullanırız. Giriş yapmak için iş mantığı **Login** widget'ının kendisinde olmalıdır. Ancak kullanıcı başarılı bir şekilde giriş yaptığında, gerçekten ana widget'ı – hatta tüm widget'ları – bilgilendirmemiz gerekir ki artık kimlik doğrulama işlemi yapılmıştır. Token, geri gönderilmelidir. Ama veriyi (durumu) ana widget'a nasıl geri iletebiliriz ki?

*İşte hile. Veriyi yukarıya geçirmeyin. Handler (işlemci) metodunu aşağıya geçirin! Dart'ta (JavaScript gibi) fonksiyonlar birinci sınıf nesnelerdir. Bu, referanslarının veri gibi geçirebileceği anlamına gelir. Bu aynı zamanda, bir fonksiyonu ana widget'tan iç widget'lara geçirebileceğiniz anlamına gelir. Artık iç widget, bu fonksiyona bir referansa sahip olduğuna göre, onu kendi fonksiyonuymuş gibi çağırabilir. Ancak elbette, iç widget onu çağırdığında, fonksiyona bir değer geçirirse, bu değer fonksiyonun tanımlandığı ana widget'ta görülecektir.*

Bu teknik, **durumu yukarıya kaldırma** (Lifting the State Up) olarak adlandırılır (Şekil 9-1).

Host.dept



Logged out

Login.dept

User

Password

\_loggedIn=false

(written host)

f(x)

***Şekil 9-1. Durumu yukarıya kaldırma***

Host.dept

|  |  |
| --- | --- |
| \_loggedIn=true  (the original)  f(x) | |
| Login.dept |  |
|  |  |
| f(x)  (A reference) | |

İç widget'a bir referans geçirin.



İç widget'ta çalıştırmak, aslında onu ana widget'ta tetikler.

### Bir durum yönetim örneği:

Bu kavramları pekiştirmek için bazı koda göz atmamız iyi olur. Diyelim ki, kullanıcının kırmızı, yeşil ve mavi değerlerini üç kaydırıcı ile ayarlayarak bir renk oluşturmasına olanak tanıyan bir uygulamamız var. Bu kaydırıcılar renkleri karıştıracak ve büyük bir dairede gösterecek (Şekil 9-2).



***Şekil 9-2. Bir örnek stateful widget***

Açıkça, büyük dairenin veri değiştikçe yeniden çizilmesi gerekiyor, bu yüzden durum (state) kullanmamız gerek. Teknik olarak, bunların hepsini tek bir büyük widget olan **ColorMixer** içinde tutabiliriz, ancak bu kitapta büyük widget'ları daha küçük, daha özelleşmiş widget'lara ayırmayı öğrendik. **ColorCircle**'ı çıkaralım ve bunu büyük daire için ve alt kısmındaki favori renkler için kullanalım. Ayrıca, etiketli üç kaydırıcımız olduğu için, hepsi aynı şeyi yaptığı için bunu da **ColorValueChanger** olarak ayıralım. Peki ya Şekil 9-3'teki düzen?



Color Mixer

One of the Sliders. We’ll reuse it three

times.

Color Circle

Color Value Changer

Draws a circle Given a size and a color.

***Şekil 9-3. Widget ağacını nasıl yerleştirebileceğimiz***

**ColorMixer** stateful olmalı:

import 'package:flutter/material.dart'; import 'ColorCircle.dart';

import 'ColorValueChanger.dart';

// The stateful widget

class ColorMixer extends StatefulWidget { ColorMixer({Key key}) : super(key: key);

\_ColorMixerState createState() => \_ColorMixerState();

}

// The state object

class \_ColorMixerState extends State<ColorMixer> {

// These three variables are the 'state' of the widget int \_red = 0;

int \_blue = 0; int \_green = 0;

@override

Widget build(BuildContext context) { return Container(

child: Column( children: <Widget>[

// This widget uses the variables (aka state) ColorCircle(color: Color.fromRGBO(\_red, \_green, \_blue, 1),

radius: 200,),

// These three pass the \_setColor function down so that the

// state \*here\* can be changed at lower levels. This is

// called "lifting state up". ColorValueChanger(property: "Red", value:\_red,

onChanged: \_setColor), ColorValueChanger(property: "Green", value:\_green,

onChanged: \_setColor), ColorValueChanger(property: "Blue", value:\_blue,

onChanged: \_setColor),

],

),

);

}

void \_setColor(String property, int value) { setState(() {

\_red = (property == "Red") ? value : \_red;

\_green = (property == "Green") ? value : \_green;

\_blue = (property == "Blue") ? value : \_blue;

});

}

}

Unutmayın, **ColorCircle**'ın ihtiyacı olan her şeyi %100 oranında ona geçiriyoruz ve bu, **ColorCircle**'ın yaşamı boyunca değişmiyor. Eğer **ColorMixer**'ın durumu değişirse, sadece **setState()** çağırarak yeniden çizilir... bu da **ColorCircle**'ı içerir. Böylece, **ColorCircle** stateless olabilir.

Aynı şekilde, her **ColorValueChanger**'a bir başlangıç değeri geçiriyoruz ve **\_setColor** metoduna bir referans veriyoruz. Hatırlayın, bir fonksiyonu aşağıya geçirmek, onu iç widget'ta kullanılabilir hale getirir ve dolayısıyla çalıştırılabilir yapar. İç widget fonksiyonu çalıştırsa da, fonksiyon aslında ana widget'ta var!

İşte iç **ColorValueChanger** widget'ında nasıl görüneceği:

@override

Widget build(BuildContext context) { return Container(

child: Column( children: <Widget>[

Text(property), Slider(

min: 0,

max: 255,

value: \_value.toDouble(), label: property, onChanged: \_onChanged,

),

],

),

);

}

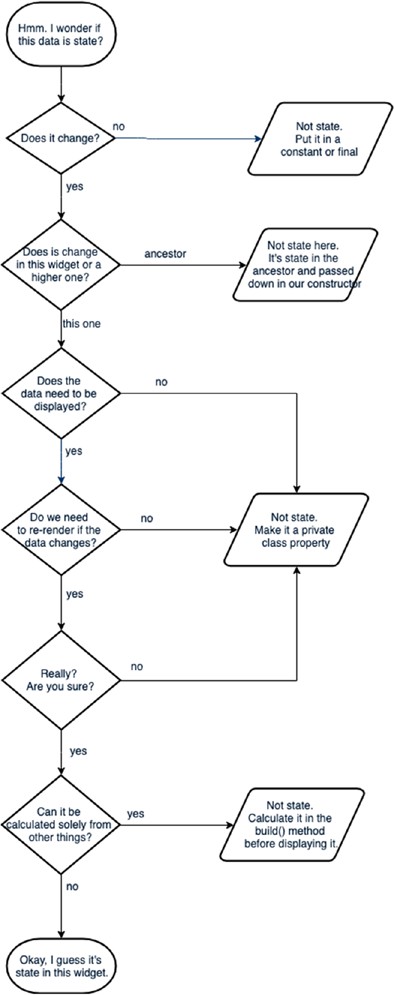
\_onChanged(double value) {

setState(() => \_value = value.round()); onChanged(property, value.round()); // Lifting state up

}

**Ne zaman durum (state) kullanmalıyız?**

Ama biliyor musunuz? Karmaşık durumu (state) önlemenin en iyi yolu, hiç durum kullanmamaktır. Neredeyse her uzman, eğer tamamen durumdan kaçınabiliyorsanız, bunu yapmanızı tavsiye eder. Ancak, ne zaman durum kullanmanız gerektiğini ve ne zaman kullanmamanız gerektiğini anlamak bazen kafa karıştırıcı olabilir.  
Örneğin, renk seçicimizdeki etiket, bileşenin içindeki bir veridir. Bu, durum (state) olmalı mı? Hayır, tabii ki hayır; çünkü değişmez. Peki ya bir for döngüsündeki sayaç? Hayır; çünkü **build()** metodunda hiçbir şeyi etkilemez, bu yüzden **setState()** içinde yer almasına gerek yoktur. Anladınız mı? Durum bazen basitleştirilebilir veya ortadan kaldırılabilir.  
**Şekil 9-4**, nasıl karar verileceğine dair bir özet sunuyor.



***Şekil 9-4. Bir widget'ta durumun (state) gerçekten kullanılması gerektiğine nasıl karar verebileceğiniz***

**İleri düzey durum yönetimi**

Bu bölümde incelediklerimiz, widget ağacı sonsuz derinliğe ulaştığında bile beklenen şekilde çalışacaktır. Ancak, uygulamanız büyüdükçe durum yönetimi giderek daha karmaşık hale geleceğini unutmayın.  
 Durum yönetimi çok karmaşık hale geldiğinde, daha gelişmiş bir durum yönetimi deseni kullanmanız daha faydalı olabilir. Bu desenler her zaman kolay öğrenilmez, ancak uygulamanız büyüdükçe, onları öğrenmek, çaba harcamaya değer hale gelir.  
Bu bölümü sindirilebilir tutmak istedik, bu yüzden bu teknolojilerin tüm detaylarına giremeyiz, ancak aynı zamanda bu problemin varmış gibi davranmak istemedik.

**Devralınan Widget**

Bu, nispeten basit bir çözüm, belki çoğu ihtiyaç için fazla basit. **InheritedWidget2** Flutter'da yerleşik bir widget'tır. Temelde, küresel değişkenlerin küçük bir setini oluşturur ve bunları ağacındaki tüm alt widget'lara kontrollü bir şekilde erişilebilir hale getirir. Diğer bazı yöntemler (ScopedModel, Provider, Redux) ise **InheritedWidget**'ın etrafında sarıcılar (wrappers) sağlar.

**Avantajlar**: Kurulması veya takibi yapılacak herhangi bir kütüphane yoktur.  
**Dezavantajlar**: **InheritedWidget** ile altındaki **StatefulWidget** arasında bazı kopyalama işlemleri vardır. Bu, üzücü bir durumdur. Ayrıca, herhangi bir veri değiştiğinde tüm alt ağaç yeniden çizilir.

#### BLoC

BLoC, **Business Logic Component** (İş Mantığı Bileşeni) için bir kısaltmadır ve bir çözümden daha çok bir tasarım desenidir. BLoC, Google tarafından oluşturulmuştur ve doğal olarak Flutter topluluğu tarafından benimsenmiştir.

2<https://docs.flutter.io/flutter/widgets/InheritedWidget-class.html>

**Avantajlar**: Topluluktaki birçok kişi size yardımcı olabilir ve yardım edecektir. Sağlam ve iyi test edilmiş bir desendir.  
 **Dezavantajlar**: Her şeyi kendiniz yazmanız gerekir; ne yerleşik bir çözüm ne de bir kütüphanedir. Bir BLoC'u nereye enjekte edeceğinizi bilmek zor olabilir.

**Kapsamlı Model**

**ScopedModel**, Brian Egan tarafından Fuchsia kod tabanından "utanmazca alınmış" bir kütüphanedir. (Hey, bunlar Brian’ın sözleri, benim değil! Kendisi mütevazı bir adam.) **ScopedModel**, dinleyicileri kaydetme yeteneğine sahip veri modelleri oluşturur. Her model, veriler değiştiğinde dinleyicilerini bilgilendirir, böylece dinleyiciler güncellenebilir. Zeki bir tasarımdır. **Avantajlar**: Sunum ve veriyi ayırma işini çok iyi yapar.  
**Dezavantajlar**: **ScopedModel**'ın, Google’ın **flutter\_provide** ile birleştirilmesi konuşuluyor, bu da durum yönetimine daha modern ve daha basit bir yaklaşım gibi görünüyor.

**Kancalar**

Paris'ten **Rémi Rousselet** tarafından React hook'larının bir implementasyonu olan **flutter\_hooks**. Artık **StatefulWidget** kullanmanıza gerek yok. Bunun yerine, **HookWidget**'ten miras alırsınız; bu widget, stateless (yani daha basit) olmasına rağmen, durum (state) değerlerini okuma ve yazma işlemi yapan özel fonksiyonlar oluşturmanıza ve bunlara erişmenize olanak tanır. Ayrıca yazmanıza gerek olmayan bazı hazır hook'larla birlikte gelir.

**Avantajlar**: Eskiden stateless olan widget'larınızı büyük ölçüde basitleştirir.  
**Dezavantajlar**: Öğrenme eğrisi. React hook'larına aşina değilseniz, nasıl çalıştığını ve kullanım kurallarını anlamak zor olabilir.

3<https://pub.dartlang.org/packages/scoped_model> 4<https://fuchsia.googlesource.com/> 5<https://pub.dartlang.org/packages/provide> 6<https://pub.dartlang.org/documentation/flutter_hooks>

#### Sağlayıcı

Bu yazının yazıldığı sırada, **Provider**, **Rémi Rousselet** tarafından yazılmış bir kütüphane ile, **Filip Hracek** ve Google'dan iyi insanlar tarafından yazılmış benzer adı taşıyan bir kütüphane arasında bazı karışıklıklar vardır. Filip, **Rémi**'nin paketinin "daha fazla özelliğe sahip" olduğunu özgürce kabul etmektedir (sic).

**Avantajlar**: Oldukça sağlam ve yetenekli bir paket olup, kullanımı diğerlerine kıyasla nispeten basittir. Yakın gelecekte, bu kütüphanenin, Redux ve/veya hook'lar konusunda önceki deneyimi olmayan geliştiriciler için ana durum yönetimi kütüphanesi haline gelmesini bekliyorum.

**Dezavantajlar**: Henüz diğer bazıları kadar popüler değil.

Not: Adlandırma nedeniyle, bu paket ile Google tarafından oluşturulan **flutter\_provide** arasında büyük bir karışıklık vardır. **flutter\_provide**, Fuchsia kod tabanından alınan ve açık kaynak olarak paylaşılan paket, **ScopedModel** ile birleştirilebilir ve deprecated (kullanımdan kaldırılabilir) olabilir.

#### Redux

Bu listedeki birkaç diğer kütüphane gibi, **Redux** da diğer teknolojilerden alınarak Dart'a taşınmış bir kütüphanedir. **Redux**, Facebook üzerinden React dünyasından gelen derin bir geçmişe sahiptir. Birkaç implementasyonu vardır, ancak en popüleri şuradadır: **flutter\_redux**. Ayrıca, üretken **Brian Egan** tarafından yazılmıştır.

**Dezavantajlar**: Çok dik bir öğrenme eğrisi vardır.  
**Avantajlar**: Çok performanslıdır. Çok ölçeklenebilirdir. Birçok React geliştiricisi zaten **Redux.js**'i biliyor. Bu nedenle öğrenme eğrisi onlar için önemli ölçüde azalır.

7<https://pub.dartlang.org/packages/provider> 8<https://github.com/google/flutter-provide/issues/3> 9<https://pub.dartlang.org/packages/flutter_redux>

**Vay! Bu çok fazla paket demek!**

Kafanız karıştı mı? Bunu suçlamıyorum. Bu paketler, aynı sorunu farklı şekillerde çözüyorlar, bazıları benzer şekilde, bazıları ise tamamen farklı stratejiler kullanıyor. Kimse, sadece dışarıda bu araçların olduğunu fark etmeniz dışında bir beklentiye sahip değil. Birinin “Durumumuz karışıyor. Belki BLoC veya ScopedModel’e bir göz atmalıyız” gibi bir şey söylediğini duyduğunuzda, en azından ne tür bir şeyden bahsettiklerini bileceksiniz. Sonra hangi teknolojiyi kullanmak isteyebileceğinizi görmek için bu araçlara daha derinlemesine bakabilirsiniz.

**Çözüm**

Bir widget'ın, içinde barındırdığı veriler aracılığıyla kendi durumunu (status) koruması gerektiği açıkça belirli zamanlar vardır. Bu durumda, buna "state" deriz ve widget'ı "stateful widget" olarak adlandırırız. Stateful widget'lar, doğası gereği stateless widget'lardan daha karmaşıktır, bu yüzden mümkünse bunlardan kaçınmaya çalışırız.  
Ayrıca, sahip olduğumuz daha fazla stateful widget ile, widget'lar arasında daha fazla durumun (state) taşınması gerekir. Bu, oldukça karmaşık hale gelebilir, bu yüzden state'i kontrol altına almak için BLoC, Redux, ScopedModel ve Provider gibi araçlar ve teknikler kullanırız.  
Neyse ki, bu kadar karmaşık temel Flutter. Sonuçta, kötü değil. Flutter uygulaması oluşturmak için gerekli her şeyi ele aldık, bu da bölümün sonunu işaret ediyor. Kitabın geri kalan kısmında, gerçek dünyada Flutter uygulamaları oluşturmanıza yardımcı olacak ek, belki de isteğe bağlı konulara geçelim!