## CHAPTER 3

**Her Şey Widget'tır (Everything Is Widgets)**

Diyelim ki sen inanılmaz yetenekli bir Lego meraklısısın ve az sayıdaki prestijli Lego Usta Yapımcılarından biri olman için teklif aldın. Tebrikler! Ayrıca, ilk görevinin 26.000 Lego parçasından oluşan, altı fit (yaklaşık 1,83 metre) boyunda bir Thor yapmak olduğunu düşünelim (Şekil 3-1).



### Şekil 3-1. Bir Lego Thor. Yazar, bu fotoğrafı bir sinema salonunda çekmiş.

© Rap Payne 2019 31

R. Payne, *Beginning App Development with Flutter*,

[https://doi.org/10.1007/978-1-4842-5181-2\_3](#_bookmark0)

Bunu nasıl yapardın? Bir dakika düşün. Devam et, biz bekliyoruz.

Hemen rastgele Lego parçalarını alıp birleştirmeye mi başlardın?  
Muhtemelen hayır. Thor’un ayak tabanlarını yerleştirip, aşağıdan yukarıya doğru mu inşa ederdin?  
Yine hayır. İşte sağduyulu bir strateji için tahminim:

1. Ne inşa edeceğini kafanda canlandırırsın. Tüm yapıyı anlamaya çalışırsın.
2. Projenin tamamının bir seferde inşa edilemeyecek kadar karmaşık olduğunu fark edersin.
3. Projeyi bölümlere ayırırsın (bacaklar, sol kol, sağ kol, gövde, sol kılıç, sağ kılıç, miğfer, pelerin, kafa).
4. Her bir bölümün hâlâ çok karmaşık olduğunu fark edersin.
5. Her bölümü alt bölümlere ayırırsın.
6. Adım 4 ve 5’i tekrar edersin, ta ki her bileşen anlaşılması, inşa edilmesi ve bakımının kolay olacağı bir seviyeye gelene kadar – hem senin hem de olası takım arkadaşların için.
7. Her basit bileşeni oluşturursun.
8. Basit bileşenleri birleştirerek daha büyük ve karmaşık bileşenler oluşturursun.
9. Adım 7 ve 8’i tekrar edersin, ta ki tüm projeyi tamamlayana kadar.

Bu sürecin bir adı var: **bileşenleştirme (componentization)** ve Flutter projelerimizde de tam olarak bu düşünme sürecini uygulayacağız.

**Bileşenleştirme (Componentization)** yeni bir şey değil. Aslında, 1968 gibi eski bir tarihte önerilmişti. Ancak bu teknik, Angular, React, Vue, Polymer ve yerel web bileşenleri gibi web framework'leri sayesinde son yıllarda popülerliğini patlama seviyesine ulaştırdı. Görünüşe göre, günümüzün "havalı" yazılımcıları hep yazılım bileşenleriyle uğraşıyor.

Karmaşık yapıları giderek daha basit parçalara ayırma fikrine **ayrıştırma (decomposition)** denir. Yazılan bu parçaları tekrar birleştirerek daha büyük bileşenler oluşturma sürecine ise **bileştirme (composition)** adı verilir.

**Flutter dünyasında, bu bileşenlere widget denir.**  
Flutter geliştiricileri sık sık **"Her şey widget'tır"** der. Bunun anlamı, sen ve ben, Google tarafından sağlanan, Flutter ile birlikte gelen widget'ları kullanacağız. Bu widget'ları bir araya getirerek kendi özel widget'larımızı oluşturacağız. Ardından, bu özel widget'ları daha karmaşık özel widget'lar oluşturmak için birleştireceğiz. Bu süreç, sonunda tam teşekküllü bir uygulama oluşturana kadar devam edecek.

**Flutter dünyasında, bileşenler widget olarak adlandırılır.**

Her uygulama iki temel bölümden oluşur:

1. **Davranış (Behavior)** – Yazılımın ne yaptığıdır. Tüm iş mantığı burada yer alır: veri okuma, yazma ve işleme işlemleri.
2. **Sunum (Presentation)** – Yazılımın nasıl göründüğüdür. Kullanıcı arayüzü, butonlar, metin kutuları, etiketler gibi bileşenler buraya dahildir.

Diğer teknolojiler bu iki kavramı farklı dillerle ele alırken, **Flutter bunları tek bir dil içinde birleştirir.**

# Kod olarak arayüz (UI as Code)

Diğer geliştirme framework'leri, bileşenleştirmenin (componentization) doğru yol olduğunu kanıtladı. Flutter ekibi ise açıkça, bu yaklaşımın büyük ölçüde benimsenmiş olduğunu belirtti.

1<http://bit.ly/componentHistory>

React'tan ilham aldıkları açıkça belirtilmiş, çünkü React da bileşenleştirmeye dayalıdır. Aslında, tüm framework yapımcıları birbirlerinden yoğun şekilde ilham alıyor gibi görünüyor. Ancak Flutter, kullanıcı arayüzünü ifade etme biçimiyle benzersizdir. Geliştiriciler, uygulamanın grafiksel kullanıcı arayüzünü ve davranışını ifade etmek için aynı Dart dilini kullanır (Tablo 3-1). Buna **“UI as code”** (Kod olarak arayüz) diyoruz.

### Tablo 3-1. Sadece Flutter, sunum ve davranış için aynı dili kullanır

|  |  |  |
| --- | --- | --- |
| **Framework** | **Davranış şu dilde ifade edilir...** | **UI şu dilde ifade edilir ...** |
| Xamarin | C# | XAML |
| React Native | JavaScript | JSX |
| NativeScript | JavaScript | XML |
| Flutter | Dart | Dart |

Peki, bu UI nasıl oluşturulur? Diğer pek çok framework ve dil gibi, bir Flutter uygulaması **main** fonksiyonu ile başlar. Flutter’da, **main** fonksiyonu bir **runApp()** fonksiyonunu çağırır. Bu **runApp()**, bir widget alır; bu widget, kök widget (root widget) olarak adlandırılabilir ve istediğiniz ismi verebilirsiniz, ancak bir Flutter **StatelessWidget** sınıfını genişleten (extend) bir sınıf olmalıdır. Görünümü şu şekildedir:

//Tüm Flutter uygulamaları için gerekli olan Dart paketini içe aktar

import 'package:flutter/material.dart';

// İşte main fonksiyonu runApp'i çağırıyor

void main() => runApp(RootWidget());

// Ve işte kök widget'ınız

class RootWidget extends StatelessWidget { @override

2Source: [https://flutter.dev/docs/resources/faq#does-flutter-come-with-](https://flutter.dev/docs/resources/faq#does-flutter-come-with-a-framework) [a-framework](https://flutter.dev/docs/resources/faq#does-flutter-come-with-a-framework)

Widget build(BuildContext context) { return Text("Hello world");

}

}

Ve işte Flutter'da bir "Merhaba dünya" uygulaması oluşturmak için gereken tek şey bu.

Ama bekleyin... bu **Text()** nedir? Bu, yerleşik bir Flutter widget'ıdır.  
Bu yerleşik widget'lar o kadar önemlidir ki, bunlara göz atmamız gerekiyor.

# Yerleşik Flutter Widget'ları (Built-in Flutter widgets)

Flutter'ın temel widget'ları, oluşturduğumuz her şeyin yapı taşlarıdır ve bunlardan tonlarcası vardır – son sayımlara göre yaklaşık 160 tane. Bu, bizim takip etmemiz için oldukça fazla widget demek. Ancak eğer bunları zihinsel olarak organize ederseniz, çok daha yönetilebilir hale gelir.

Bu widget'lar ana kategorilere ayrılır:  
• **Değer widget'ları**  
• **Yerleşim widget'ları**  
• **Navigasyon widget'ları**  
• **Diğer widget'lar**

Not: Bunlar, Flutter'ın resmi kategori listesi değildir. Flutter'ın 14 kategorisi burada listelenmiştir: [https://flutter.dev/docs/](https://flutter.dev/docs/development/ui/widgets)

[development/ui/widgets](https://flutter.dev/docs/development/ui/widgets). Sadece bunları yeniden organize etmenin, onları daha net bir şekilde anlamamıza yardımcı olduğunu düşündük.

Hepsinin listesini burada bulabilirsiniz:: [https://flutter.dev/docs/reference/](https://flutter.dev/docs/reference/widgets) [widgets](https://flutter.dev/docs/reference/widgets)

Bu kategorilerin her birine bir veya iki örnekle kısa bir göz atacağız, sonra ilerleyen bölümlerde daha derinlemesine incelemeler yapacağız. İlk olarak değer widget'larıyla başlayalım.

# Değer Widget'ları (Value widgets)

Bazı widget'lar bir değer tutar, belki bu değerler yerel depolamadan, internetteki bir hizmetten ya da kullanıcının kendisinden gelmiştir. Bu widget'lar, kullanıcıya değerleri göstermek ve kullanıcıdan gelen değerleri uygulamaya almak için kullanılır.

Önemli bir örnek **Text widget'ı**dır; bu widget, bir miktar metin görüntüler. Bir diğer örnek ise **Image widget'ı**dır; bu widget, bir .jpg, .png veya başka bir resmi görüntüler.

İşte birkaç tane daha değer widget'ı:

Checkbox CircularProgressIndicator Date & Time Pickers DataTable DropdownButton FlatButton FloatingActionButton FlutterLogo

Form

FormField Icon IconButton Image

LinearProgressIndicator PopupMenuButton Radio

RaisedButton RawImage

RefreshIndicator RichText

Slider Switch Text TextField Tooltip

Değer widget'larını bir sonraki bölümde daha ayrıntılı olarak inceleyeceğiz..

# Yerleşim Widget'ları (Layout widgets)

Yerleşim widget'ları, sahnemizin doğru bir şekilde yerleşmesini sağlamak için bize büyük bir kontrol sunar – widget'ları yan yana veya üst üste yerleştirme, kaydırılabilir hale getirme, sarmasını sağlama, widget'ların etrafındaki boşluğu belirleyerek sıkışık hissettirmemelerini sağlama ve daha fazlası:

Align AppBar AspectRatio Baseline BottomSheet ButtonBar Card

Center Column

ConstrainedBox Container CustomMultiChildLayout Divider

Expanded ExpansionPanel

FittedBox Flow

FractionallySizedBox GridView IndexedStack IntrinsicHeight IntrinsicWidth LayoutBuilder LimitedBox

ListBody ListTile ListView MediaQuery

NestedScrollview OverflowBox

Padding PageView Placeholder Row Scaffold Scrollable Scrollbar

SingleChildScrollView SizedBox SizedOverflowBox SliverAppBar SnackBar

Stack Table Wrap

Bu, büyük bir konudur ve bu nedenle kendisine özel bir bölüm ayırdık: Bölüm 6, "Widget'larınızı Yerleştirme".

# Navigasyon Widget'ları (Navigation widgets)

Uygulamanız birden fazla sahneye (“ekran,” “sayfa,” ne isterseniz) sahipse, bunlar arasında geçiş yapmanın bir yoluna ihtiyacınız olacak. İşte bu noktada **Navigasyon widget'ları** devreye girer. Bu widget'lar, kullanıcınızın bir sahneyi görmesini ve sonra bir sonraki sahneye geçmesini kontrol eder. Genellikle bu, kullanıcı bir düğmeye dokunduğunda yapılır.

Bazen navigasyon düğmesi, bir sekme çubuğunda veya ekranın sol tarafından kayan bir çekmecede bulunur. İşte bazı navigasyon widget'ları:

AlertDialog BottomNavigationBar Drawer

MaterialApp Navigator SimpleDialog

TabBar TabBarView

Bu widget'ların nasıl çalıştığını, Bölüm 7, "Navigasyon ve Yönlendirme"de öğreneceğiz

# Diğer Widget'lar (Other widgets)

Ve hayır, tüm widget'lar bu düzenli kategorilere girmez. Geri kalanları bir **çeşitli widget'lar** kategorisinde toplayalım. İşte bazı çeşitli widget'lar:

GestureDetector Dismissible

Cupertino Theme

Transitions Transforms

Bu çeşitli widget'ların birçoğu, kitabın ilerleyen bölümlerinde, doğal olarak uyacakları yerlerde ele alınmıştır. **GestureDetector** o kadar önemli bir widget'tır ki, kendi bölümüne sahip olmuştur: Bölüm 5, "Gesturlere Yanıt Verme".

# Kendi StatelessWidget'larınızı nasıl oluşturursunuz? (How to create your own stateless widgets)

Biliyoruz ki, bu yerleşik widget'ları birleştirerek kendi özel widget'larımızı oluşturacağız ve bunlar, diğer yerleşik widget'larla birleştirilerek sonunda bir uygulama oluşturacak.

Widget'lar ustaca tasarlanmıştır çünkü her bir widget kolayca anlaşılabilir ve bu yüzden kolayca bakım yapılabilir. Widget'lar dışarıdan soyutlanmışken, içeride mantıklı ve tahmin edilebilir şekilde çalışır. Onlarla çalışmak bir rüya gibidir.

Her widget, özellikler ve metotlar içerebilen bir sınıftır. Her widget, sıfır veya daha fazla parametreye sahip bir yapıcıya (constructor) sahip olabilir. En önemlisi, her widget'ın bir **build** metodu vardır. Bu metod, bir **BuildContext** alır ve tek bir Flutter widget'ı döndürür. Eğer bir widget'ın neden böyle göründüğünü merak ediyorsanız, onun **build** metodunu bulabilirsiniz.

:

**BuildContext**'e dikkatiniz dağılmasın. Bu, framework tarafından kullanılır ve bazen buna başvururuz, ancak bu örnekleri kitabın ilerleyen bölümlerine saklayacağız. Şu an için, sadece bunu bir özel widget yazmanın tarifi olarak düşünün.

class RootWidget extends StatelessWidget { @override

Widget build(BuildContext context) { return Text('Hello world');

}

}

Bu "merhaba dünya" örneğinde, bölümün başlarında tekrar ettiğimiz gibi, bir **Text** widget'ı görüntülüyoruz (Şekil 3-2). Tek bir iç widget çalışır, ancak gerçek dünya uygulamaları çok daha karmaşık olacaktır. Kök widget, birçok başka alt widget'tan oluşabilir:

class FancyHelloWidget extends StatelessWidget { Widget build(BuildContext context) {

return MaterialApp( home: Scaffold(

appBar: AppBar(

title: Text("A fancier app"),

),

body: Container(

alignment: Alignment.center, child: Text("Hello world"),

),

floatingActionButton: FloatingActionButton( child: Icon(Icons.thumb\_up),

onPressed: () => {},

),

),

);

}

}



***Şekil 3-2. Bu basit widget ile oluşturulan uygulama***

Gördüğünüz gibi, **build** metodu tek bir widget döndürüyor, bir **MaterialApp**, ancak bu widget bir **Scaffold** içeriyor ve bu **Scaffold**, üç alt widget içeriyor: bir **AppBar**, bir **Container** ve bir **FloatingActionButton** (Şekil 3-3). Her biri, sırasıyla kendi alt-alt widget'larını içeriyor.

Malzeme Uygulaması

İskelet Yapı

Uygulama Çubuğu

Text

Kapsayıcı

Text

Yüzen Eylem Düğmesi

Icon

***Şekil 3-3. Yukarıdaki örnek uygulamamızdan widget ağacı***

Build metodunuz her zaman bu şekilde çalışacaktır. Tek bir, büyük, iç içe geçmiş ifade döndürecektir. Widget'lar, widget'lar içinde, widget'lar içinde, size kendi ayrıntılı özel widget'ınızı oluşturma imkanı sağlar.

## Widget'ların Anahtarları (Keys) vardır (Widgets have keys)

Flutter hakkında konuşan diğer geliştiriciler, sanal DOM’dan bahsedebilirler. Bu, React dünyasından gelir. (Unutmayın ki, Flutter, React’in mükemmel mimarisinden büyük ölçüde ilham almıştır.) Teknik olarak, Flutter’ın bir DOM'u yoktur, ancak ona benzer bir şey tutar – **eleman ağacı**. Eleman ağacı, ekrandaki tüm widget'ların küçük bir kopyasıdır. Flutter, mevcut bir eleman ağacını ve üzerine toplu değişiklikler uygulanmış bir başka ağacı tutar.

Flutter, her küçük değişikliği ekrana uygular ve ardından bunu saniyede yüzlerce kez yeniden render etmeye çalışsaydı gerçekten çok yavaş olurdu. Bunun yerine, Flutter tüm bu değişiklikleri eleman ağacının bir kopyasına uygular. Ardından periyodik olarak, mevcut eleman ağacını değiştirilmiş olanla karşılaştırarak (diff) gerçekten neyin yeniden render edilmesi gerektiğine karar verir. Sadece gereken kısımlar yeniden render edilir. Bu çok daha hızlıdır.

Ancak bazen Flutter, eleman ağaçlarındaki widget'ları eşleştirirken karışabilir. Veriler değiştiğinde ve widget'lar yanlış konumda çizildiğinde, veriler ekranda güncellenmediğinde veya kaydırma pozisyonunuz korunmadığında, programatik olarak anahtar (key) atamanız gerektiğini anlayacaksınız.

Çoğu zaman anahtarlar hakkında endişelenmenize gerek yoktur. O kadar nadir gereklidir ki, şunları anlamanız yeterlidir:

1. Anahtarlar vardır ve Flutter’ın bunlara neden ihtiyaç duyabileceğini bilmelisiniz.
2. Eğer widget'larınız veriler değiştiğinde beklediğiniz şekilde yeniden çizilmiyorsa, anahtarlar bu sorunları çözebilir.
3. Belirli widget'lara anahtar atama fırsatınız vardır.

Eğer bu konu şu an için sizi tatmin etmiyorsa, büyük Emily Fortuna, anahtarlar hakkında süper on dakikalık bir video kaydetmiştir.

## Widget'inize bir değer geçirme (Passing a value into your widget)

## Bu formülün ne anlama geldiğini biliyor musunuz?

## y = f(x)

## Matematik bölümü öğrencileri bunu "Y, X'in bir fonksiyonudur." olarak tanıyacaktır. Bu, X (bağımsız değişken) değiştikçe, Y (bağımlı değişken) 'nin öngörülebilir bir şekilde değişeceğini kısaca ifade eder. Flutter da bu fikir üzerine kurulu, ancak Flutter'da formül şu şekilde okunur:

## Scene = f(Data)

## Diğer bir deyişle, uygulamanızdaki veri değiştikçe, ekran da buna göre değişecektir. Ve siz, geliştirici olarak, bu verinin nasıl sunulacağını, widget'larınızda build metodu yazarken karar verirsiniz. Bu, Flutter'ın temel bir konseptidir.

## Veri nasıl değişebilir? İki yol vardır:

## Widget, dışarıdan yeni veri ile yeniden render edilebilir.

## Veri, belirli widget'lar içinde saklanabilir.

## İlkini konuşalım. Veri bir widget'a geçmek için, bunu bir constructor parametresi olarak geçirebilirsiniz, örneğin şöyle:

## Widget build(BuildContext context) {

## return Person("Sarah"); // "Sarah"yı widget'a geçirme

## }

## Bir widget bir kişiyi nasıl render edeceğini temsil ediyorsa, yukarıda yaptığımız gibi bir firstName (ilk ad) geçirmek çok normal bir şey olur. Bunu yaparsanız, widget'ınızın constructor'ını bu değeri alacak şekilde yazmanız gerekecektir:

## class Person extends StatelessWidget {

## final String firstName;

## Person(this.firstName);

## Widget build(BuildContext context) {

## return Text('$firstName');

## }

## }

## Bu Dart sözdizimi. Üç şey dikkat edilmesi gereken nokta var. İlk olarak, constructor'da input parametresini listeleyeceksiniz ("this.firstName" örneğindeki gibi). İkinci olarak, "this." ifadesini önüne koyduğunuzdan emin olun. "this." ifadesi, bunu class düzeyindeki bir property ile eşleştirir, constructor fonksiyonunda yerel bir parametreyle değil. Ve üçüncü olarak, karşılık gelen class property'sini final olarak işaret edin.

## İki ya da daha fazla property geçmek isterseniz, şöyle olur:

## Widget build(BuildContext context) {

## return Person("Sarah", "Ali");

## }

## Tabii ki, iki değer geçirmek, iki final değişkeni ve bunları ele alacak iki constructor parametresi oluşturmanız gerektiği anlamına gelir:

## class Person extends StatelessWidget {

## final String firstName;

## final String lastName;

## Person(this.firstName, this.lastName);

## Widget build(BuildContext context) {

## return Text('$firstName $lastName');

## }

## }

## Tahmin edebileceğiniz gibi, bunlar konumsal olarak eşleştirilir, bu da kolayca karışabileceğiniz ve çok esnek olmayan bir durum olabilir. Daha iyi bir uygulama, adlandırılmış parametreler kullanmaktır:

## Widget build(BuildContext context) {

## return Person(firstName: "Sarah", lastName: "Ali");

## }

## Bu, widget'ınızı kullanan diğer geliştiriciler için karışıklığı azaltır.

## Widget'ınızı bu değeri alacak şekilde yazma şekliniz:

## class Person extends StatelessWidget {

## final String firstName;

## final String lastName;

## Person({this.firstName, this.lastName});

## Widget build(BuildContext context) {

## return Container(child: Text('$firstName $lastName'));

## }

## }

## Farkı görüyor musunuz? Çok ince bir fark var. Constructor parametrelerinin etrafında artık süslü parantezler var. Bu, onları isteğe bağlı ve adlandırılmış yapar.

## İpucu: Yukarıdaki üç örneğin tamamında, kullanmakta olduğunuz Person sınıfının aynı dart dosyasına tanımlanmış olduğunu varsayıyoruz. Ancak daha iyi bir uygulama, her sınıfı ayrı bir dart dosyasına koymak ve bu dosyayı başka dart dosyalarına import etmektir.

import 'Person.dart';

# Stateless and Stateful widgets

Şimdiye kadar, **StatelessWidget** oluşturmak için çaba harcadık. Bu yüzden **StatefulWidget**'in de olduğunu tahmin etmişsinizdir. Haklısınız. **StatelessWidget**, kendi durumunu tutmayan bir widget’tır. **StatefulWidget** ise durumunu tutar.

Bu bağlamda "Durum" (state), widget’ın yaşamı boyunca değişebilen veriyi ifade eder. Önceki örneğimizdeki **Person** widget'ını düşünün. Eğer sadece kişinin bilgilerini görüntüleyen bir widget ise, bu stateless olmalıdır. Ancak kullanıcıya **TextField** aracılığıyla verileri değiştirme imkanı sunduğumuz bir **person maintenance** widget'ı yapıyorsak, o zaman **StatefulWidget** kullanmamız gerekir.

StatefulWidget'lar hakkında ilerleyen bölümlerde bir bölüm bulunmakta. Eğer şimdiden daha fazla bilgi almak isterseniz, kitabın 9. bölümünde yer alan **"Managing State"** kısmına göz atabilirsiniz. Sonrasında buraya geri dönebilirsiniz.

.

## O zaman hangisini oluşturmalıyım? (So which one should I create?)

Kısa cevap, stateless widget oluşturun. Bir stateful widget kullanmanız gerektiği zaman haricinde asla kullanmayın. Oluşturduğunuz tüm widget'ların stateless olacağını varsayın ve onları bu şekilde başlatın. Gerçekten state'e ihtiyacınız olduğundan emin olduğunuzda stateful widget'lara dönüştürün. Ancak, state'in geliştiricilerin düşündüğünden daha sık engellenebileceğini fark edin. Widget'ları daha basit hale getirebilmek, yazmayı, bakımını yapmayı ve genişletmeyi kolaylaştırmak için state'den kaçının. Takım arkadaşlarınız bunun için size teşekkür edecektir.

Not: Aslında üçüncü bir widget türü daha vardır, o da InheritedWidget. Bir değeri InheritedWidget içinde belirlersiniz ve herhangi bir alt widget, bu değeri doğrudan almak için yukarıya doğru ağacın içinde erişebilir. Bu biraz ileri seviye bir konu, ancak Rémi Rousselet bana başımı derde sokardı eğer bunu bahsetmeseydim. Daha fazla bilgi için 9. Bölüm “State Yönetimi”ni okuyabilir veya Emily Fortuna’nın InheritedWidget üzerine kısa özet videosunu izleyebilirsiniz.

: <http://bit.ly/inheritedWidget>.

# Sonuç (Conclusion)

Şimdi, Flutter uygulamalarının tamamen widget'lar üzerine kurulu olduğunu biliyoruz. Kendi özel Stateless veya Stateful widget'larınızı oluşturacak ve her biri bir build yöntemiyle yerleşik Flutter widget'larının bir ağacını render edecek. Bu nedenle, yerleşik Flutter widget'ları hakkında bilgi sahibi olmamız gerektiği açık ve bunu bir sonraki bölümde öğreneceğiz.