**What’s Revolutionary about Flutter**

**What is Flutter?**

UPDATE: [Flutter is now in beta](https://flutter.io/). [Read the announcement](https://medium.com/flutter-io/announcing-flutter-beta-1-build-beautiful-native-apps-dc142aea74c0).

[fuente](https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514)

The Flutter mobile app SDK is a new way to build *fast*, *beautiful* mobile apps that helps developers break away from “cookie cutter” apps that have been so common in the past. People who try Flutter really like it; for example, see [this](https://medium.com/@aubykhan/how-i-built-a-cross-platform-connected-app-in-7-days-93728a987424), [this](https://medium.com/@germansaprykin/flutter-first-steps-fffd906f093b), or [this](https://code.tutsplus.com/tutorials/developing-an-android-app-with-flutter--cms-28270). Or here’s a [list of articles and videos](https://github.com/Solido/awesome-flutter) compiled by a third party.

As with any new system, people want to know what makes Flutter different, or put another way, “what is new and exciting about Flutter?” That is a fair question, and this article will try to answer it from a technical viewpoint — not just *what* is exciting, but *why*.

But first, a little history.

**A brief history of mobile app development**

Mobile app development is a relatively recent field of endeavor. Third-party developers have been able to build mobile apps for less than a decade, so it is no surprise that tools are still evolving.

**OEM SDKs**

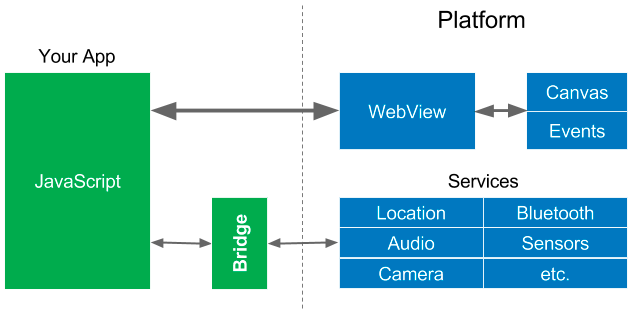
The Apple iOS SDK was released in 2008 and the Google Android SDK in 2009. These two SDKs were based on different languages: Objective-C and Java, respectively.



Your app talks to the platform to create widgets, or access services like the camera. The widgets are rendered to a screen canvas, and events are passed back to the widgets. This is a simple architecture, but you pretty much have to create separate apps for each platform because the widgets are different, not to mention the native languages.

**WebViews**

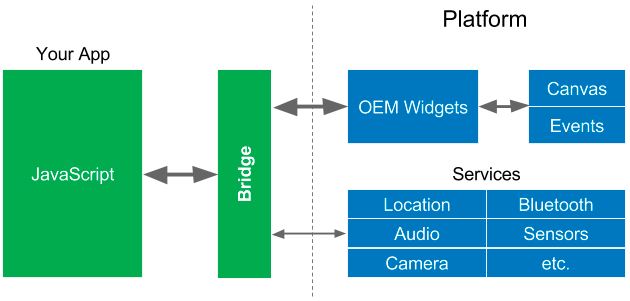
The first cross-platform frameworks were based on JavaScript and WebViews. Examples include a [family of related frameworks](https://en.wikipedia.org/wiki/Apache_Cordova): PhoneGap, Apache Cordova, Ionic, and others. Before Apple released their iOS SDK they encouraged third party developers to build webapps for the iPhone, so building cross-platform apps using web technologies was an obvious step.



Your app creates HTML and displays it in a WebView on the platform. Note that it is difficult for languages like JavaScript to talk directly to native code (like the services) so they go through a “bridge” that does context switches between the JavaScript realm and the native realm. Because platform services are typically not called all that often, this did not cause too many performance problems.

**Reactive Views**

Reactive web frameworks like [ReactJS](https://facebook.github.io/react/) (and [others](https://github.com/Raynos/mercury)) have become popular, mainly because they simplify the creation of web views through the use of programming patterns borrowed from [reactive programming](https://en.wikipedia.org/wiki/Reactive_programming). In 2015, React Native was created to bring the many benefits of reactive-style views to mobile apps.



React Native is very popular (and deserves to be), but because the JavaScript realm accesses the OEM widgets in the native realm, it has to [go through the bridge](https://medium.com/@talkol/performance-limitations-of-react-native-and-how-to-overcome-them-947630d7f440) for those as well. Widgets are typically accessed quite frequently (up to 60 times a second during animations, transitions, or when the user “swipes” something on the screen with their finger) so this can cause performance problems. As [one article about React Native](https://medium.com/@talkol/performance-limitations-of-react-native-and-how-to-overcome-them-947630d7f440) puts it:

*Here lies one of the main keys to understanding React Native performance. Each realm by itself is blazingly fast. The performance bottleneck often occurs when we move from one realm to the other. In order to architect performant React Native apps, we must keep passes over the bridge to a minimum.*

**Flutter**

Like React Native, Flutter also provides reactive-style views. Flutter takes a different approach to avoiding performance problems caused by the need for a JavaScript bridge by using a compiled programming language, namely [Dart](https://www.dartlang.org/). Dart is compiled “ahead of time” (AOT) into native code for multiple platforms. This allows Flutter to communicate with the platform without going through a JavaScript bridge that does a context switch. Compiling to native code also improves app startup times.

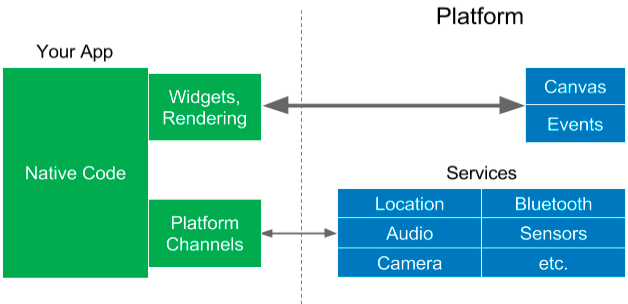
The fact that Flutter is the only mobile SDK that provides reactive views without requiring a JavaScript bridge should be enough to make Flutter interesting and worth trying, but there is something *far more revolutionary*about Flutter, and that is how it implements *widgets*.

**Widgets**

Widgets are the elements that affect and control the view and interface to an app. It is not an overstatement to say that the widgets are one of the most important parts of a mobile app. In fact, widgets alone can *make* or *break* an app.

* The **look** and **feel** of widgets is paramount. Widgets need to look good, including on various screen sizes. They also need to feel natural.
* Widgets must perform **fast**:to create the widget tree, inflate the widgets (instantiating their children), lay them out on the screen, render them, or (especially) animate them.
* For modern apps, widgets should be **extensible**and**customizable**. Developers want to be able to add delightful new widgets, and customize all widgets to match the app’s brand.

Flutter has a new architecture that includes widgets that look and feel good, are fast, and are customizable and extensible. That’s right, Flutter does not use the OEM widgets (or DOM WebViews), it provides its own widgets.



Flutter moves the widgets and the renderer from the platform into the app, which allows them to be customizable and extensible. All that Flutter requires of the platform is a canvas in which to render the widgets so they can appear on the device screen, and access to events (touches, timers, etc.) and services (location, camera, etc.).

There is still an interface between the Dart program (in green) and the native platform code (in blue, for either iOS or Android) that does data encoding and decoding, but this can be orders of magnitude faster than a JavaScript bridge.

Moving the widgets and the renderer into the app does affect the size of the app. The minimum size of a Flutter app on Android is approximately 6.7MB, which is similar to minimal apps built with comparable tools. It is up to you to decide if the benefits of Flutter are worth any tradeoff, so the rest of this article discusses these benefits.

**Layout**

One of the biggest improvements in Flutter is how it does layout. **Layout**determines the size and position of widgets based on a set of rules (also called constraints).

Traditionally, layout uses a large set of rules that can be applied to (virtually) any widget. The rules implement multiple layout methods. Let’s take as an example CSS layout because it is well known (although layout in Android and iOS is basically similar). CSS has properties (the rules), which are applied to HTML elements (the widgets). CSS3 defines 375 properties.

CSS includes a number of layout models, including (multiple) box models, floating elements, tables, multiple columns of text, paged media, and so on. Additional layout models, like flexbox and grid, were added later because developers and designers needed more control over layout and were using tables and transparent images to get what they wanted. In traditional layout new layout models cannot be added by the developer, so flexbox and grid had to be added to CSS and implemented on all browsers.

Another problem with traditional layout is that the rules can interact (and even conflict) with each other, and elements often have dozens of rules applied to them. This makes layout slow. Even worse, layout performance is typically of order N-squared, so as the number of elements increases, layout slows down even more.

Flutter started as an experiment performed by members of the Chrome browser team at Google. We wanted to see if a faster renderer could be built if we ignored the traditional model of layout. After a few weeks, we had achieved significant performance gains. We discovered:

* Most layout is relatively simple, such as: text on a scrolling page, fixed rectangles whose size and position depend only on the size of the display, and maybe some tables, floating elements, etc.
* Most layout is local to a subtree of widgets, and that subtree typically uses one layout model, so only a small number of rules need to be supported by those widgets.

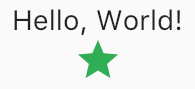
We realized that layout could be simplified significantly if we *turned it on its head*:

* Instead of having a large set of layout rules that could be applied to any widget, each widget would specify its own simple layout model.
* Because each widget has a much smaller set of layout rules to consider, layout can be optimized heavily.
* To simplify layout even further, we turned almost everything into a widget.

Here’s Flutter code to create a simple widget tree with layout:

new Center(  
 child: new Column(  
 children: [  
 new Text('Hello, World!')),  
 new Icon(Icons.star, color: Colors.green)  
 ]  
 )  
)

This code is semantic enough that you can easily imagine what it will produce, but here’s the resulting display:



In this code [everything is a widget](https://flutter.io/widgets/), including the layout. The Center widget centers its child inside its parent (for example, the screen). The Column layout widget arranges its children (a list of widgets) vertically. The column contains a Text widget and an Icon widget (which does have a property, its color).

In Flutter, centering and padding are widgets. Themes are widgets, which apply to their children. And even applications and navigation are widgets.

Flutter includes quite a few widgets for doing layout, not just columns but also rows, grids, lists, etc. In addition, Flutter has a unique layout model we call the “sliver layout model” which is used for scrolling. Layout in Flutter is so fast it can be used for scrolling. Think about that for a moment. Scrolling must be so instantaneous and smooth that the user feels like the screen image is attached to their finger as they drag it across the physical screen.

By using layout for scrolling, Flutter can implement advanced kinds of scrolling, like these shown below. Note that these are animated GIF images, Flutter is even smoother. You can (and should!) run these apps yourself; see the Resources section at the end of this article.

Flutter Gallery app

Posse Gallery app

Posse Gallery app

Most of the time, Flutter can do layout in a single pass, which means in linear time, so it can handle large numbers of widgets. Flutter also does caching and other things so it can avoid doing layout at all, when possible.

**Custom design**

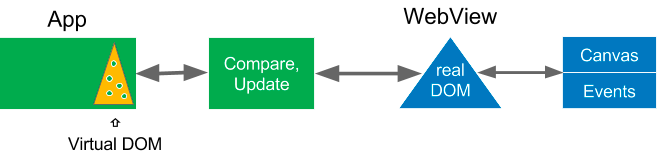
Because widgets are now part of the app, new widgets can be added and existing widgets can be customized to give them a different look or feel, or to match a company’s brand. The trend in mobile design is away from the cookie cutter apps that were common a few years ago and toward custom designs that delight users and win awards.

Flutter comes with rich, customizable widget sets for Android, [iOS](https://flutter.io/widgets/cupertino/), and [Material Design](https://material.io/) (in fact, we have been told that Flutter has one of the highest fidelity implementations of Material Design). We used the customizability of Flutter to build these widget sets, to match the look and feel of native widgets on multiple platforms. App developers can use the same customizability to further tweak widgets to their wants and needs.

**More about Reactive Views**

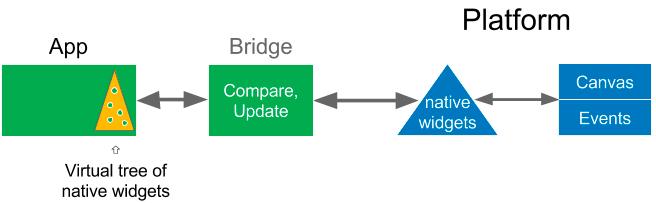
Libraries for reactive web views introduced **virtual DOM**. DOM is the HTML Document Object Model, an API used by JavaScript to manipulate an HTML document, represented as a tree of elements. Virtual DOM is an abstract version of the DOM created using objects in the programming language, in this case JavaScript.

In reactive web views (implemented by systems like [ReactJS](https://facebook.github.io/react/) and others) the virtual DOM is immutable, and is rebuilt from scratch each time anything changes. The virtual DOM is compared to the real DOM to generate a set of minimal changes, which are then executed to update the real DOM. Finally, the platform re-renders the real DOM and paints it into a canvas.



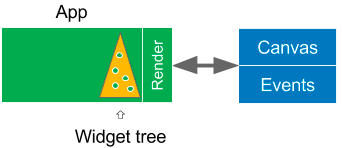
This may sound like an awful lot of extra work, but it is well worth it because [manipulating the HTML DOM is very expensive](https://www.codecademy.com/articles/react-virtual-dom).

React Native does a similar thing, but for mobile apps. Instead of DOM, it manipulates the native widgets on the mobile platform. Instead of a virtual DOM, It builds a virtual tree of widgets and compares it to the native widgets and only updates those that have changed.



Remember that React Native has to communicate with the native widgets through the bridge, so the virtual tree of widgets helps keep passes over the bridge to a minimum, while still allowing the use of native widgets. Finally, once the native widgets are updated, the platform then renders them to the canvas.

React Native is a big win for mobile development, and was an inspiration for Flutter, but Flutter takes this a step further.



Recall that in Flutter, the widgets and the renderer have been lifted up out of the platform into the user’s app. There are no native OEM widgets to manipulate, so what was a virtual widget tree is now *the* widget tree. Flutter renders the widget tree and paints it to a platform canvas. This is nice and simple (and fast). In addition, animation happens in user space, so the app (and thus the developer) have more control over it.

The Flutter renderer itself is interesting: it uses several internal tree structures to render only those widgets that need to be updated on the screen. For example, the renderer uses “[structural repainting using compositing](https://youtu.be/UUfXWzp0-DU)” (“structural” meaning *by widget*, which is more efficient than doing it by rectangular areas on the screen). Unchanged widgets, even those that have moved, are “[bit blitted](https://en.wikipedia.org/wiki/Bit_blit)” from cache, which is super fast. This is one of the things that makes scrolling so performant in Flutter, even in advanced scrolling (discussed and shown above).

For a closer look at the Flutter renderer, I [recommend this video](https://youtu.be/UUfXWzp0-DU). You can also look at the code, because [Flutter is open source](https://github.com/flutter). And of course, you can customize or even replace the whole stack, including the renderer, compositor, animation, gesture recognizer, and (of course) the widgets.

**The Dart programming language**

Because Flutter — like other systems that use reactive views — refreshes the view tree for every new frame, it creates many objects that may live for only one frame (a sixtieth of a second). Fortunately, Dart uses “[generational garbage collection](https://en.wikipedia.org/wiki/Tracing_garbage_collection#Generational_GC_.28ephemeral_GC.29)” that is very efficient for these kind of systems, because objects (especially short-lived ones) are relatively cheap. In addition, allocation of objects can be done with a single pointer bump, which is fast and doesn’t require locks. This helps avoid UI [jank](https://en.wiktionary.org/wiki/jank) and stutter.

Dart also has a “[tree shaking](https://medium.com/@Rich_Harris/tree-shaking-versus-dead-code-elimination-d3765df85c80)” compiler, which only includes code that you need in your app. You can feel free to use a large library of widgets even if you only need one or two of them.

For more information about Dart, read “[Why Flutter uses Dart](https://hackernoon.com/why-flutter-uses-dart-dd635a054ebf)”.

**Hot reload**

One of the most popular features of Flutter is its fast, stateful hot reload. You can make a change to a Flutter app while it is running, and it will reload the app’s code that has changed and let it continue from where it left off, often in less than a second. If your app encounters an error, you can typically fix the error and then continue on as if the error never happened. Even when you have to do a full reload, it is fast.

hot stateful reload

Developers tell us that this lets them “paint” their app, making one change at a time and then seeing the results almost instantly, without having to restart the app.

**Compatibility**

Because widgets (and the renderer for those widgets) are part of your app, not the platform, no “[compat libraries](https://developer.android.com/topic/libraries/support-library/index.html)” are needed. Your apps will not only work, but they will work the same on recent OS versions — Android Jelly Bean and newer and iOS 8.0 and newer. This significantly reduces the need to test apps on older OS versions. Plus it is likely that your apps will work on future OS versions.

There is one potential concern that we get asked about. Because Flutter doesn’t use the OEM native widgets, will it take long for the Flutter widgets to be updated when a new version of iOS or Android comes out that supports a new kind of widget, or changes the look or behavior of an existing widget?

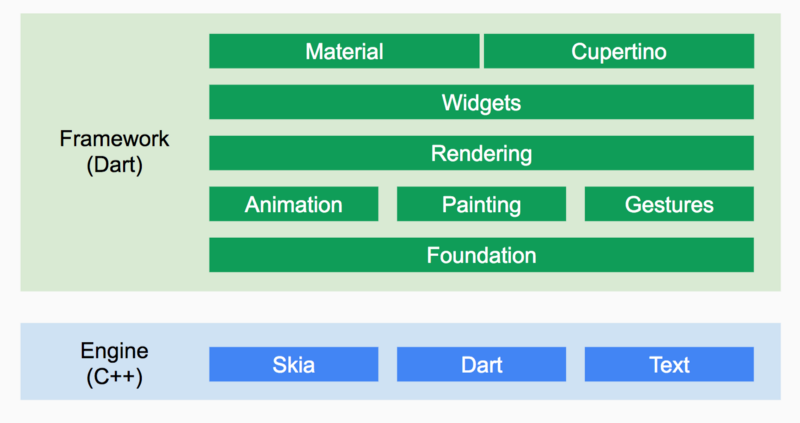
* First of all, Google is a big internal user of Flutter, so we have a strong incentive to update the widget sets to keep them current and as close to the current OEM widgets as possible.
* If there is ever a time when we are too slow in updating a widget, Google isn’t the only user of Flutter with an incentive to keep the widgets current. Flutter’s widgets are so extensible and customizable that anyone can update them, even you. One doesn’t even have to file a pull request. You will never have to wait for Flutter itself to be updated.
* And the above points apply only if you want to have the new change reflected in your app. If you *don’t* want a change to affect the way your app looks or behaves, you’re good. Widgets are part of your app, so a widget will never change out from under you and make your app look bad (or worse, break your app).
* As an added benefit, you can write your app so it uses the new widget even on older OS versions.

**Other benefits**

Flutter’s simplicity makes it fast, but it is the pervasive customizability and extensibility that makes it powerful.

Dart has a [repository of software packages](https://pub.dartlang.org/flutter/packages/) so you can extend the capabilities of your apps. For example, there are a number of packages that make it easy to [access Firebase](https://github.com/flutter/plugins/blob/master/FlutterFire.md) so you can build a “serverless” app. An outside contributor has created a package that lets you access a [Redux data store](https://pub.dartlang.org/packages/flutter_built_redux). There are also packages called “[plugins](https://flutter.io/developing-packages/)” that make it easier to access platform services and hardware, such as the accelerometer or the camera, in an OS-independent way.

Of course, Flutter is also [open source](https://github.com/flutter), which coupled with the fact that the Flutter rendering stack is part of your app, means that you can customize almost anything you want for an individual app. Everything in green in this figure can be customized:



**So, “What’s new and exciting about Flutter?”**

If anyone asks you about Flutter, now you know how to answer them:

* The advantages of reactive views, with no JavaScript bridge
* Fast, smooth, and predictable; code compiles AOT to native (ARM) code
* The developer has full control over the widgets and layout
* Comes with beautiful, customizable widgets
* Great developer tools, with amazing hot reload
* More performant, more compatibility, more fun

Did you notice what I left off this list? It is something that is usually the first thing people mention when they talk about Flutter, but to me it is one of the least interesting things about Flutter.

It is the fact that Flutter can build beautiful and fast apps for multiple platforms from a single codebase. Of course, that should be a given! It is customizability and extensibility that makes it easy to target Flutter to multiple platforms without giving up performance or power.

**Revolutionary**

I also never fully explained why Flutter is “revolutionary”. It just seems fitting, because one of the [first major apps built with Flutter by outside developers](https://blog.goposse.com/a-revolutionary-app-for-a-revolutionary-show-1b687f24023b) is the official app for “[Hamilton: An American Musical](http://www.hamiltonbroadway.com/)”, which takes place around the time of the American Revolutionary War. [Hamilton is one of the most popular](http://www.cnn.com/2017/08/14/entertainment/hamilton-app/index.html) Broadway musicals of all time.



The agency, [Posse](http://goposse.com/), says they picked Flutter because they needed to build the app “[in only three short months](https://developers.googleblog.com/2017/08/hamilton-app-takes-stage.html)”. They call it “A revolutionary app for a revolutionary show” and say “Flutter is an excellent choice for beautiful, high-performance, brand-driven mobile experiences.” [They also gave a talk](https://youtu.be/prlK_QL_qOA) at the Google Developer Days conference on their experience building an app with Flutter. The app is available on [Android](https://play.google.com/store/apps/details?id=com.hamilton.app) and [iOS](https://itunes.apple.com/us/app/hamilton-the-official-app/id1255231054), and [received rave reviews](http://www.cnn.com/2017/08/14/entertainment/hamilton-app/index.html).

**Join the Revolution!**

Flutter is currently in beta release. We are still adding more features to it, and we have more optimizations planned. However, groups both inside and outside of Google are already using it to build mission-critical apps.

If you are interested in Flutter, [you can install it](https://flutter.io/setup/) and [play around with](https://flutter.io/getting-started/) some sample apps that come with the installation. Be sure to check out the stateful hot reload.

If you aren’t a developer or just want to see some apps, you can install apps built with Flutter and see how they look and perform. I recommend the [Hamilton app](http://www.hamiltonbroadway.com/#newyork-app), but there are others. You should also watch the [video from Google I/O](https://goo.gl/csEr1f) where they live code a Flutter app.

**Resources**

* Watch the [videos from DartConf 2018](https://www.youtube.com/results?search_query=DartConf+2018), held January 23–24, 2018 in Los Angeles, California
* [Introducing Flutter video](https://youtu.be/fq4N0hgOWzU).

**Websites:**

* The [Flutter website](https://flutter.io/)
* The [source repositories](https://github.com/flutter/flutter) (pull requests welcome!)
* [More helpful links](https://github.com/Solido/awesome-flutter)
* [Gitter channel](https://gitter.im/flutter/flutter)

**Videos:**

* [Experience building the Hamilton app, at GDD](https://youtu.be/prlK_QL_qOA)
* [Live coding a Flutter app, at Google I/O](https://goo.gl/csEr1f)
* [Live coding with a designer, at Google I/O](https://goo.gl/N29fiK)
* “[Flutter, a new hope](https://goo.gl/g1tDQR)” at Droidcon Italy
* “[The Flutter rendering pipeline](https://goo.gl/RH3N4Y)”

**Apps:**

* The app for [Hamilton: An American Musical](http://www.hamiltonbroadway.com/#newyork-app)
* Flutter Gallery: [for Android](https://play.google.com/store/apps/details?id=io.flutter.gallery), [on Github](https://github.com/flutter/flutter/tree/master/examples/flutter_gallery) (also included in the flutter install at examples/flutter\_gallery)
* [Posse Gallery](https://github.com/flutter/posse_gallery)