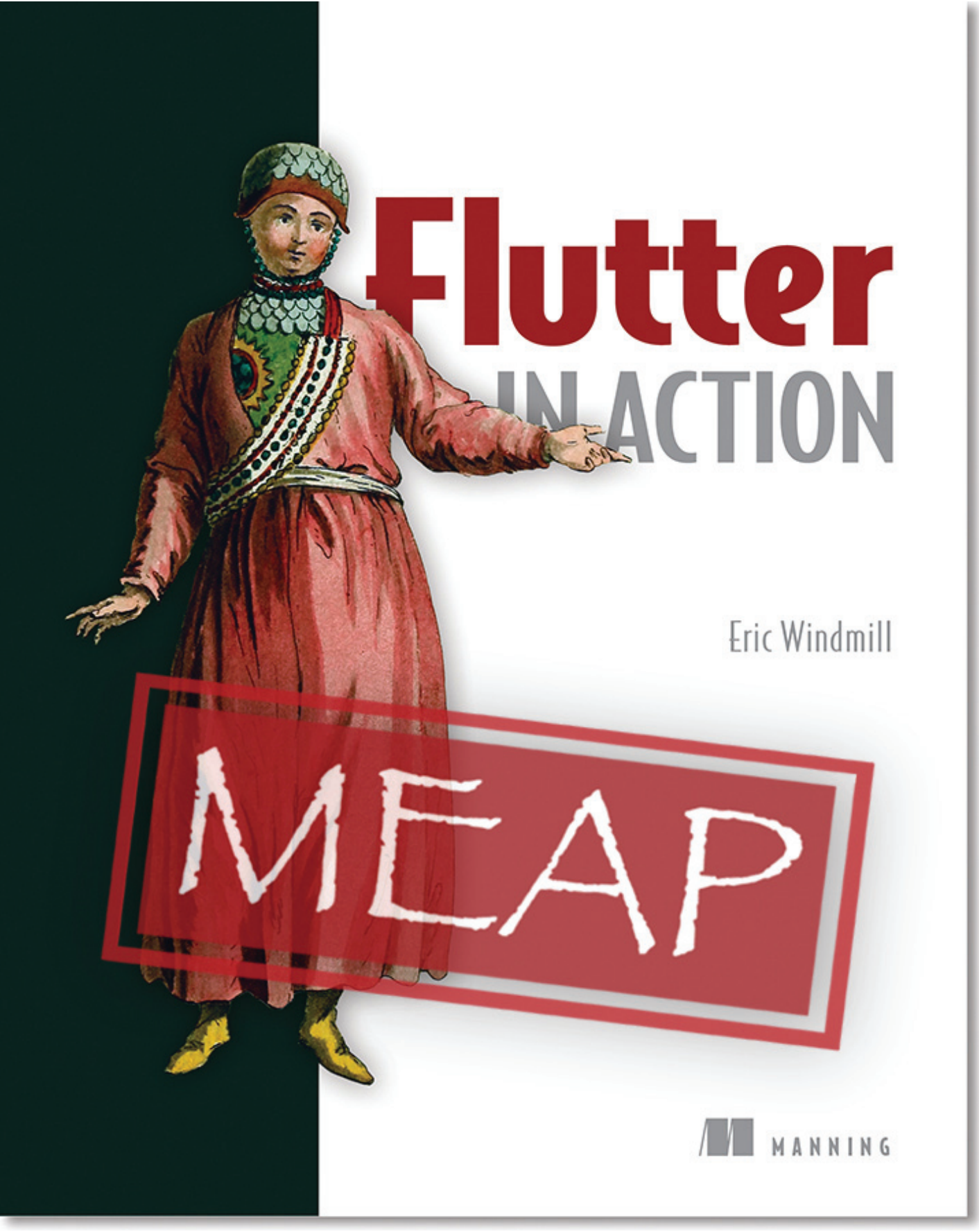

Flutter
IN ACTION
Eric Windmill
MEAP
MANNING

**MEAP Edition**
**Manning Early Access Program**
# Flutter in Action
**Version 4**



For more information on this and other Manning titles go to
manning.com

# *welcome*

Thank you for putting your faith in *Flutter in Action*. This book is about making mobile apps for smart phones. Today, 'smart phones' are just called 'phones', and we hardly every use them to talk. In fact, its estimated that we spend almost 90% of time on mobile devices looking at apps.

Humans, as it turns out, are entitled, emotional, hard to please, and all around more complicated than computers and software. Software is elegant, predictable, and (unfortunately), hard to make well. It's our job to take on this hard task, and deliver beautiful products to the people with impossible standards in ivory towers. Like the artist, we'll never be able to produce something that's up to these standards. Unlike the artist, millions of people will still insist on using our work, despite complaining about it. This tenacity and dedication is what makes us heroes.

Like all heroes, we must be resourceful and smart. The tools we choose to leverage can set us apart. John Henry was just a man without his hammer. The Greeks may have never won the (mythological) war with Troy without their Trojan Horse. Yuri Gagarin changed the landscape of human possibility because he had the best technology.

That's why Flutter is important. It's the first tool that lets us create applications with native -quality experiences for both major platforms at astonishing speed. We can leverage this tool to be the best mobile developers that we can be.

When it comes to software, quality and speed of development are often at odds with other. Flutter challenges this fact of human nature though. Flutter uses a highly productive language and boasts sub-second hot reload in development mode. This means that you can slow down, breath, and write quality code and still produce apps at an amazing velocity.

When I started using Flutter in September 2017, it was in alpha. I started using it because my boss told me to. I had no opinions on it because I had never heard of it. But, I got hooked immediately. Not only is the end product of the highest quality, but the development process is perhaps the most enjoyable of any SDK that I've used. The tooling, the community, the API, and the Dart language are all a joy to participate in.

Since I started using Flutter, it's grown quite a bit. There are no shortage of resources to learn Flutter. My goal with this book, however, is to cover the whole shebang in one go. You'll want to have a bit of experience making applications in the past, be-it web or mobile applications, because there's going to be lots of jargon that you'll need to know. Other than that, though, I'll cover it all. You'll learn about Dart, and you'll learn about Flutter. And (even if you don't care), you'll learn about how Flutter works under the hood. By the end of the book, you'll have a bit of experience sharing business logic between Flutter and web applications.

Perhaps most importantly, though, it’s my goal that you’ll become a better developer. Tools come and go, but best practices are rooted in history. If you're comfortable with any sort of modern application programming, you'll get to reinforce some skills, and maybe see a different approach.

This book will cover a lot of ground. But, I'll walk you through every step. By the end of this, I hope we'll both be better at what we do. If you have any questions, comments or suggestions, please post them in the Author Online forum. Feedback is essential to progress, so don't be shy!

—Eric Windmill

# *brief contents*

# 1 Introducing Flutter

**In this chapter:**

- What is Flutter?
- What is Dart?
- When is Flutter the right tool?
- Why does Flutter use Dart?
- The benefits of Flutter
- Who uses Flutter

Flutter is a mobile sdk that, at its core, is fundamentally about empowering everyone to build beautiful mobile apps. Whether you come from the world of web development or native mobile development, Flutter makes it easier than ever to create mobile apps in a familiar, simplified way — without ever giving up control to the framework.

In fact, that's how I've found myself here, writing this book. I had to learn Flutter because of my job, and I loved it from the moment I started. I effectively became a mobile developer overnight, because Flutter felt so familiar to my web development background. (The Flutter team has openly stated being influenced by ReactJS.)

Flutter isn't only about being easy, though. It's also about control, and that's where 'interested, but skeptical' turns into 'excited'. You can build exceptional mobile apps using Flutter with a shallow knowledge of the framework. But, you can also create incredible and unique features, if you so choose, because Flutter exposes *everything* to the developer.

This is a book about writing a (relatively) small amount of code and getting back a

fully featured, beautiful mobile app. In the grand scheme, mobile app development is new. It can be a pain point for developers and companies alike. I believe Flutter has changed that. That is a hill I'm willing to die on.

This books has one goal: to turn you into a (happy) mobile developer.

## 1.1 On Dart

Besides explaining Flutter in depth, I will also introduce the basics of Dart. Dart is a programming language. And programming languages are, as it turns out, hard to learn. The fundamentals of Dart are similar to all high-level languages. You'll find familiarity in Dart syntax if you're coming from JavaScript, Java or any other 'C-like' language. You'll feel comfortable with Dart's object oriented design if you're coming from Ruby or Python.

Like all languages, though, the devil is in the details (and as they say, doubly in the bubbly). The joys of Dart and the complexity of writing good Dart code lies not the syntax, but in the pragmatics.

There's good news, though. Dart excels at being a 'safe' language to learn. Google didn't setout to create anything innovative with Dart. They simply wanted to make a language that was simple, prductive and could be compiled into JavaScript. In the next chapter, I will go deeper into Google's motivation for building Dart.

There is nothing particularly exciting about its syntax, and no special operators that will throw you through a loop. In Dart (unline JavaScript), there is one way to say true: True. There is one way to say false: False.

`If (3) {` would make Dart blow up, but it's coersed to `true` in JavaScript.

Dart is, at its core, a productive, predictable, and simple language.

That said, take the time you need to learn Dart. Writing an app in Flutter is simply writing Dart. Flutter is, underneath it all, a library of Dart classes. There is no markup language involved or JSX style hybrid language. It'll be much easier to be a productive Flutter developer if you're comfortable writing effective Dart code. I'll cover Dart in-depth in the next chapter.

## 1.2 Why Does Flutter Use Dart?

If you're coming literally any other background (and you're like me), you've probably complained about the fact that Flutter uses Dart, and not x language for y reason. (Developers are, believe it or not, opinionated.)

- Besides the fact that Dart isn't *your* favorite language, there are reasons to be skeptical of this choice. It's not one of the hot languages of today. It's not even one of the top 50 most used languages. What gives? Is Google just using it because it's their language? I'd imagine that played a role, but there are practical reasons, too.
- Dart supports both Just In Time (JIT) compiling and Ahead of Time(AOT)

compiling.
    - The AOT compiler changes Dart into efficient native code. This makes Flutter fast (a win for the user and the developer), but it also means that almost all of the framework is written in Dart. For you, the developer, that means you customize almost everything.
    - Dart's optional Just-In-Time compiling allows hot-reloading to exist. Fast development and iteration is a key to the joy of using Flutter.
- Dart is Object Oriented. This makes it easy to write visual user-experiences exclusively with Dart, with no need for a markup language.
- Dart is a productive and predictable language. It's easy to learn and it feels familiar. Whether you come from a dynamic language or a static language, you can get up and running with ease.

## 1.3 Who Uses Flutter?

At the time of writing, Flutter is used by big and small companies alike in production. I've been lucky enough to use Flutter at work since September 2017, when the technology was still in its alpha stage. By the time you read this, Flutter will be in (at least) version 1.0.0, and my company will have migrated all of our clients off of our native apps and onto our Flutter app.

While this isn't a book about me, I am going to tell you a bit about what I do, because I want you to know that I'm confident in the future of Flutter. A large number of my career eggs are in the Flutter basket.

The company that I work for (as of summer 2018) is in the enterprise space. Our product is used by some big companies like Stanford University, Wayfair and Taylor Parts. We're building a BYOD (bring your own database) platform that lets customers plug in a few options, press a few buttons, and it spits out a mobile and web app to manage workflows and business-y enterprise issues. Our mobile app supports offline usage, esri maps, and real-time feedback. We've done this all with Flutter (on mobile).

The point is this: don't be afraid of the limitations of cross-platform tools.

We aren't the only ones using Flutter in production. As of writing this, Google AdWords and Alibaba are both using Flutter in production. You can see more examples of who's using Flutter (including the app I've worked on) on Flutter's website on the showcase page.

### 1.3.1 Teams, Project Leads, and CTOs

Flutter has proved, in front of my very eyes, to increase productivity and collaboration by orders of magnitude. Before Flutter, everytime a new feature was introduced to our product, it had to be written and maintained three times by three different teams. Three different teams that could hardly collaborate because they had different skill sets.

Flutter solved that problem. Our three teams (web, iOS, and Android) are now one unified clients team. We all have the same skill set and we can now all collaborate and

lend helping hands.

### 1.3.2 Individual Developers

As developers, we often get starry-eyed and want to start a new project that will change everything. The key to success with this sort of work is busting out the project quickly. I can't count how many times I started a new project and quit before I began because of JavaScript build tools and set up. If you need to build an MVP fast, and iterate quickly, Flutter just works.

### 1.3.3 Code School Students and Recent CS Grads

Code schools are quite popular, and unfortunately for the graduates, that means that there are many grads fighting for the same Junior level jobs. My advice to anyone looking for their first job is to build a portfolio that sets you apart. Having a published mobile app with actual users will do just that, and it's easier than ever to achieve that with Flutter.

### 1.3.4 Open Source Developers

Flutter is open source. Dart is open source. The tools and the libraries are open source.

### 1.3.5 People Who Value Speed

Flutter is for people who want to build an app quickly that doesn't sacrifice performance. By speed, I mean the speed at which you can write code and iterate, and the speed at which Flutter builds. Thanks to hot-reloading, Flutter rebuilds your application in sub-second time as you're developing.

I would also argue that Dart makes you more productive, adding more speed. Dart is strictly typed and fully featured. Dart will save you from having to solve problems that are already solved, and the syntax and tooling makes debugging a breeze.

### 1.3.6 People Who Are Lazy

I'm a lazy developer. If a problem is solved, I don't want to waste time solving it again. Flutter comes with a massive library of Material Design widgets that are beautiful and ready to use out of the box. I don't have to worry myself with design and building complicated pieces of a mobile app (a la navigation-drawer). I want to focus on the business logic that makes my app unique.

### 1.3.7 People Who Value Control

Although I'm lazy, I do want to know that if I need to, I can change anything about my app. Flutter exposes everything the developer. If you need to write some custom rendering logic, you can do that. You can take control of animations between frames. Every high-level widget in Flutter is a thread that can be unspooled and followed to the inner-workings of the framework.

## 1.4 Who is This Book For?

This book assumes that you've developed an application before. That could be a web app, a native mobile app, Xamarin, or something I don't even know about. The important thing for you to understand is how a modern application work. I don't expect you to know how to write code across the whole stack, only that you know what a modern stack consists of. This book will focus on writing a mobile application in Flutter, and will throw around common terms like 'state', 'store', 'services', etc.

If you meet the above criteria, I can assume that you're familiar with the common threads across all programming languages. You don't need to know Dart, but you do need to know about basic data structures (i.e. Map, Lists, etc) and features of all high-level languages (control-flow, loops, etc).

Finally, this book assumes that you know some high-level information about software engineering in general. For example, Dart and Flutter operate completely in the camp of the Object Oriented paradigm. You don't need to be an Object Oriented Design guru, but you should be aware of it.

Also, It's okay if you've never written a line of JavaScript in your life, but in order to understand Dart, it would be helpful if you know JavaScript's (original) primary use and what makes it special. (Basically, that it runs in browsers and manipulates the DOM).

This book is perfectly suitable for you if you're a Jr. Developer, a Sr. Developer or anywhere in between. The prerequisites are simply that you've written code before and you're interested in learning Flutter.

## 1.5 Other Mobile Development Options

Before I offer up unsolicited opinions on your other options, I want to make this crystal clear. Good developers think critically about which tools and technologies should be used in every different situation. And, Flutter is not the answer 100% of the time.

But, this book is about convincing you otherwise.

### 1.5.1 Native Development (iOS and Android)

Your first choice is to write native apps for iOS and Android. This gives you maximum control, debugging tools, and (potentially) a very performant app. At a company, this likely means you have to write everything twice; once for each platform. You likely need different developers on different teams with different skill sets that can't easily help each other.

### 1.5.2 Cross-Platform JavaScript Options

Your second option: cross-platform, JavaScript based tools such as web views and React Native. These aren't bad options, either. The problems you experience with native development disappear. Every front-end web developer on your team can chip in and help, all they need is some modern JavaScript skills. This is precisely why large

companies such as AirBnb, Facebook, and Twitter have used React Native on core products.

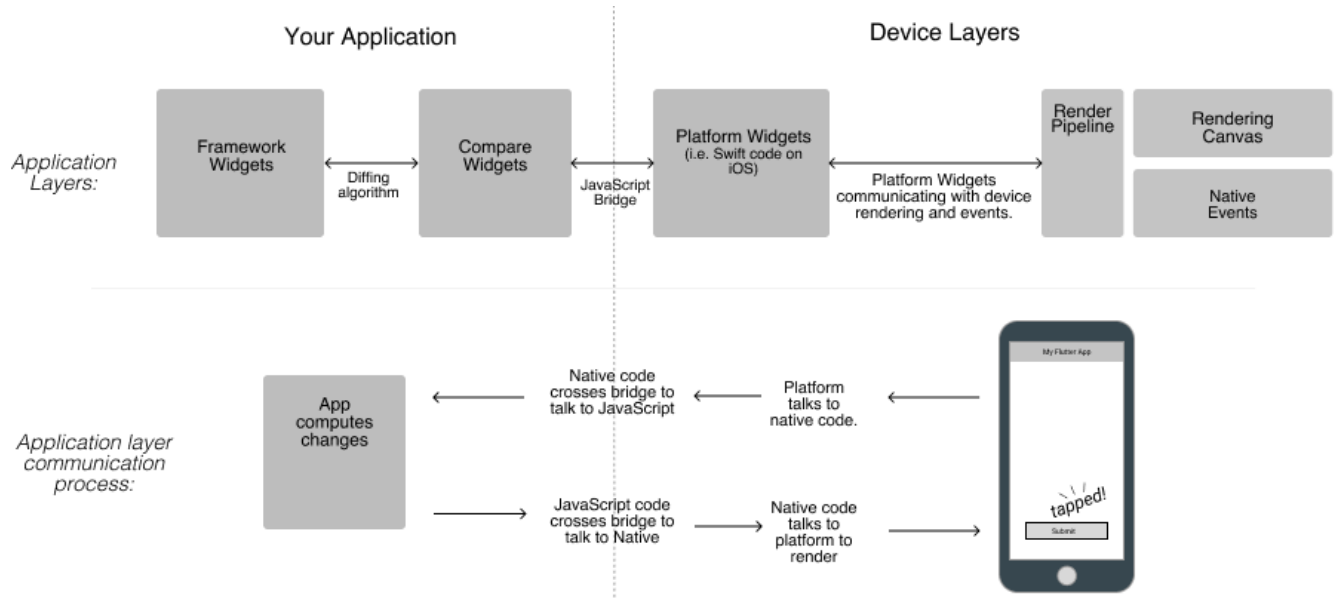
Of course, there are some drawbacks. (You knew there would be drawbacks.) The biggest of which is referred to as the ‘JavaScript bridge’.

The first 'mobile apps' to be built cross platform are simply web views that run on Webkit (a browser rendering engine). These are literally just embedded web pages. The problem with this is basically that manipulating the DOM is very expensive and doesn’t perform well enough to make a great mobile experience.

Some platforms have solved this problem by building the 'JavaScript bridge'. This bridge let’s javascript talk directly to native code.

This is much more performant than the webviews, because you elimate the DOM from the equation, but it’s still not ideal. Everytime your app needs to talk directly to the rendering engine, it has to be compiled to native code to 'cross the bridge'. On a single interaction, the bridge must be crossed *twice*, once from platform to app, and then back from app to platform.

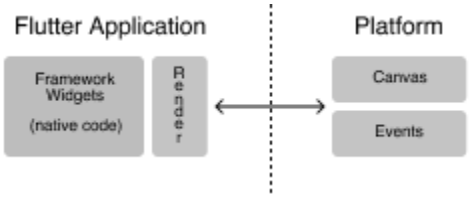
**Figure 1.1. The JavaScript bridge is a major bottleneck in JavaScript to mobile frameworks. The JavaScript isn’t compiled to native code, and must therefor must compile 'on the fly' while the app is running.**




The difference between JavaScript platforms and Flutter is that Flutter compiles directly to ARM code when it’s built for production.(ARM is the processor used in modern mobile devices, wearables, IoT devices, etc.) And, Flutter ships with its own rendering engine. Rendering engines are outside the scope of this book (and my

knowledge, for that matter). Importantly and simply, though, these two factors mean that your app is running natively, and doesn't need to cross any bridge. It talks *directly* to native events and controls every pixel on the screen *directly*.

**Figure 1.2. The Flutter platform, in the context of the JavaScript bridge.**




The JavaScript bridge is a marvel of modern programming, to be sure, but it presents two big problems.

One: debugging is hard. When there's an error in the runtime compiler, that error has to be traced back across the JS bridge and found in the JavaScript code. It may be in markup or css-like syntax as well. The debugger itself may not work as well as we'd like it to.

The bigger issue, though, is performance. The JavaScript bridge is very expensive. Everytime something in the app is tapped, that event must be sent across the bridge to your JavaScript app. The result, for lack of better term, is 'jank'.

Many of these cross-platform problems are solved with Flutter. Later in this chapter, I'll show you how.

## 1.6 The Immediate Benefits of Flutter

I'm going to make an assumption about you. Since you're reading this book (and this section), it follows that you're curious about Flutter. It's also likely that you're skeptical.

I admire how thorough you are in vetting your options.

The reasons you have for being skeptical are fair. It's a new technology. That means breaking changes in the API. It means missing support for important features (such as Google Maps). It seems possible that Google could abandon it altogether one day.

And, (despite the fact that you believe Dart is great language), that doesn't change the fact that Dart isn't widely used, and many third-party libraries that you want may not exist.

Now that I've pinned you, let me change your mind. The API will likely not change, as the the company that's developing Flutter uses it internally on major revenue generating apps such Google AdWords.

Dart has recently moved into version 2, which means it will be a long time until it changes much. It will likely be years until breaking changes are introduced, which in

computer world is practically forever.

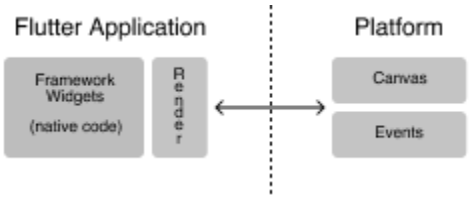
Finally, there are indeed missing features, but Flutter gives you the complete control to add your own native plugins. And in fact, many of the most important operating system plugins already exist, such as a Map Plugin, Camera, Location services, and device storage.

### 1.6.1 No JavaScript Bridge

Other cross platform frameworks rely on the JavaScript bridge between your application code and your operating system's code to function. This is a major bottleneck in development and in your application's performance. It leads to, for lack of a better term, jank. Scrolling isn't smooth, it's not always performant, and it's hard to debug.

Flutter compiles to actual native code and is rendered using Skia. The app itself is running in native, so there's no reason to convert Dart to native. This means that it doesn't lose any of the performance or productivity when it's running on a user's device.

**Figure 1.3. Flutter compiles to native code, which makes is as fast as native apps on their respective platforms.**




### 1.6.2 Compile Time

If you're coming from native development, one of your major pains is the development cycle. iOS is infamous for it's insane compile times. In Flutter, a full compile generally takes less than 30 seconds, and incremental compiles are subsecond thanks to hot-reload. At my day job, we develop features for our mobile client first because Flutter's development cycle allows us to move so quickly. Only when we're sure of our implementation do we go write those features in the web client.

### 1.6.3 Write Once, Test Once, Deploy Everywhere

Not only do you get to write your app one time and deploy to iOS and Android, you also only have to write your tests once. Dart unit testing is quite easy, and Flutter includes a library for testing Widgets.

### 1.6.4 Code Sharing

I'm going to be fair here: I suppose this is technically possible in JavaScript as well. But, it's certainly not possible in native development. With Flutter and Dart, your web

and mobile apps can share all the code except each client's views. (Of course, only if you're using Dart for your web apps.) You can quite easily use dependency injection to run an AngularDart app and Flutter app with same models and controllers.

And obviously, even if you don't want to share code between your web app and your mobile app, you're sharing all your code between the iOS and Android apps.

In practical terms, this means that you are super productive. I mentioned that we develop our mobile features first at my day job. Because we share business logic between web and mobile, once the mobile feature is implemented, we only have to write views for the web that expect that same controller data.

### 1.6.5 Productivity and Collaboration

Gone are the days of separate teams for iOS and Android. In fact, whether your use JavaScript in your web apps or Dart, Flutter development is familiar enough that all your teams will be unified. It's not a stretch by any means to expect a JavaScript web developer to also effectively develop in Flutter and Dart. If you believe me here, then it follows that your new unified team will be three times more productive.

### 1.6.6 Code Maintenance

Nothing is more satisfying then fixing a bug once and having it corrected on all your clients. Only in very specific cases is there a bug on the Flutter produced iOS app and not the Android one (and vice versa). In 100% of these cases, these bugs aren't bugs, but cosmetic issues because Flutter follows device OS design systems in it's built in widgets. Because these are issues like text-size or alignment, they are trivial in the context of using engineering time to fix.

### 1.6.7 The Bottom Line: Is Flutter for You?

You can spend all day listening to people tell you the benefits or downfalls of any technology. At the end of the day, though, its all about the tradeoff. So *should you care about Flutter*? I've sprinkled in the answer to this above, but let me give it to you straight:

Are you an **individual developer** working on a side project new product? Then the answer is simple: yes. This is absolutely for you. The amount of time you'll spend getting up to speed with Dart and Flutter will pay off big time in the long run.

Are you a **CTO** deciding if your company should adopt the technology? Well, this is a little more nuanced.

If you're starting a new project and trying to leverage the skills of web developers, then absolutely. You'll get better performance and a more cohesive team, and all your developers (mobile and web), will be able to pick it up fast.

However, if you have a big team of iOS and Android engineers, then probably not. If you have the resources to not be concerned with keeping parity between your clients, then why re-write them? Why gamble on a new technology? Flutter is about

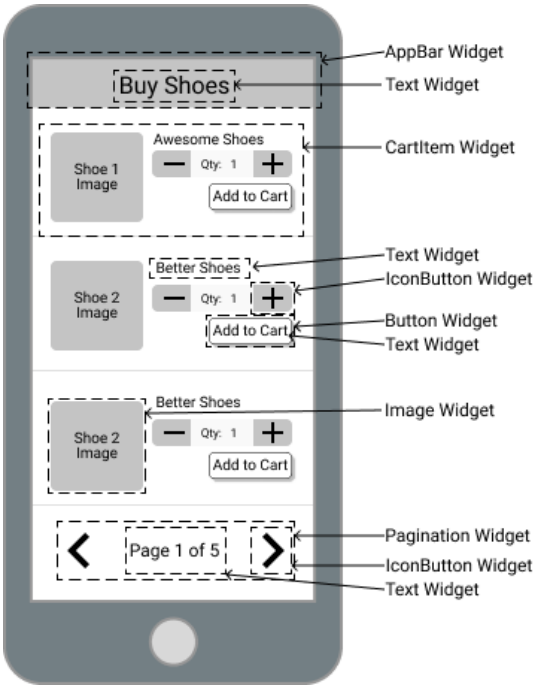
empowering anyone to build near-native quality apps, but if you're *already* empowered to build native apps, it's probably not for you. This is essentially why AirBnb abandoned React Native.

## 1.7 A Brief Intro into How Flutter Works

At a high level, Flutter is a reactive rendering library, much like ReactJS on the web. In a nutshell, you build a mobile UI by composing together a bunch of smaller components called **Widgets**. Everything is a widget, and Widgets are just Dart classes that know how to describe their view.

Suppose your building this shopping cart app. Everything is widgets inside widgets inside widgets.

**Figure 1.4. Everything is a widget. Some are highlighted here with dashed boxes, but in reality there are far more.**




Some widgets have state, for example the quantity widgets that keep track of how many of each item to add to the cart.

When a widget's state changes, the framework is alerted and it compares the new widget tree description against the previous description, and changes only the widgets that are necessary.

Looking at our cart example, when a user presses the '+' button on the quantity widget,

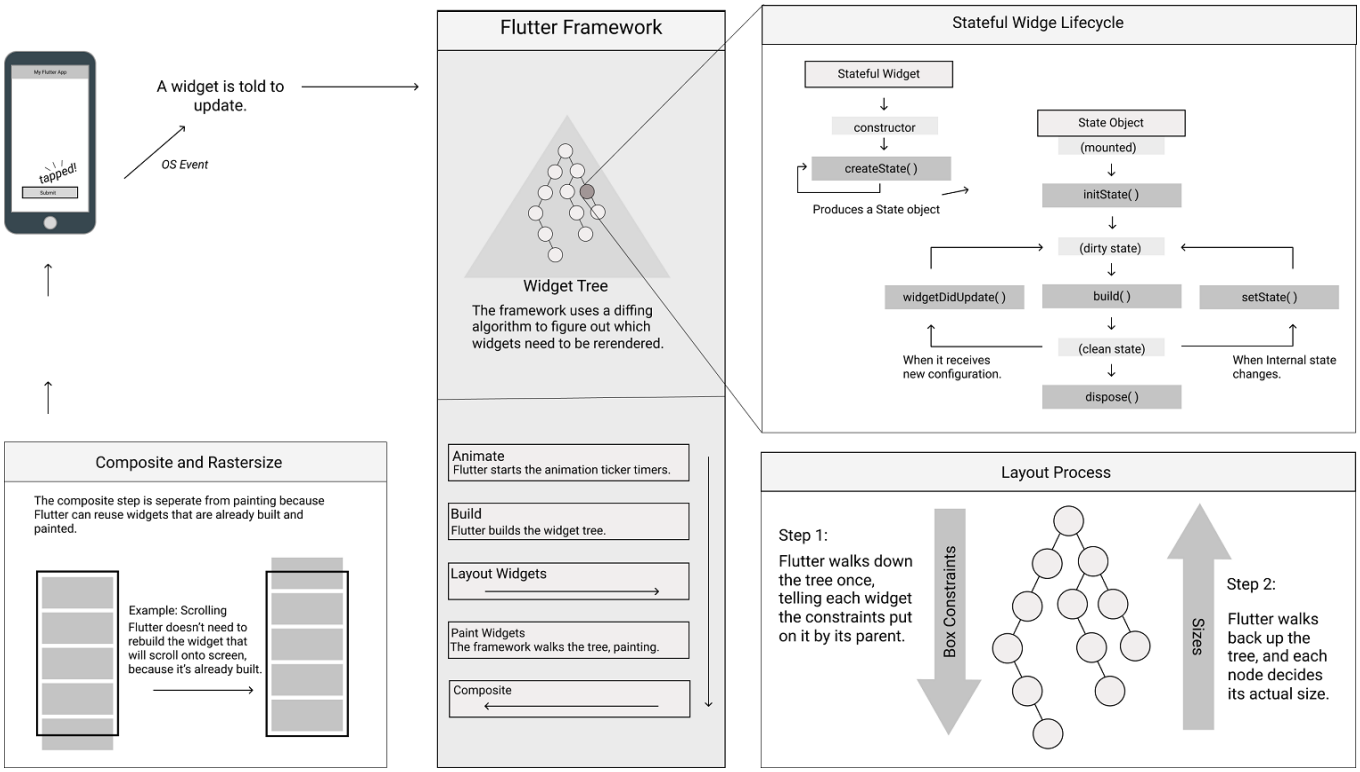
it updates internal state, which tells Flutter to repaint all widgets who depend on that state.(In this case, the text widget)

**Figure 1.5. When button (a) is pressed, icon (b) is repainted.**


Those two ideas (widgets and updating state) are truly the core of what we care about as developers. For the remainder of this chapter, let's break down, in-depth, what's *really* happening.

I'll start by explaining what *we* care about: widgets and state. Then I'll walk through how Flutter actually handles our code when an application is running.

**Figure 1.6. Flutter Framework Model**



### 1.7.1 Everything is A Widget

This is a core idea in Flutter. Literally everything is a widget. And a widget is the only object model that Flutter knows about. There aren't separate views and controllers and models. Its widgets all the way down.

As a developer, you will spend most of your Flutter time writing widgets, much like in a ReactJS app you write components.

A widget can define any aspect of an application's view. Some widgets, such as the `Row`, define aspects of layout. Some are less abstract and define structural elements, like `Button` and `TextField`. Even the root of your application is a Widget.

Using the shopping cart example again, there are a lot more widgets than we can 'see', because they define layout, styles, animations, etc.

**Figure 1.7. Examples of layout widgets. There are far more here, in reality.**

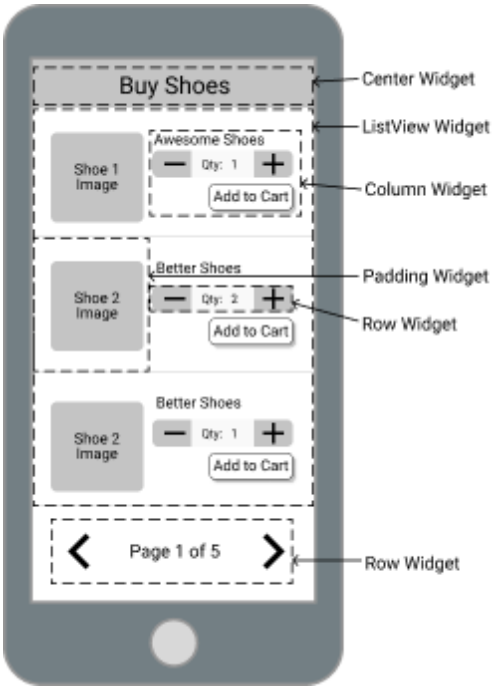
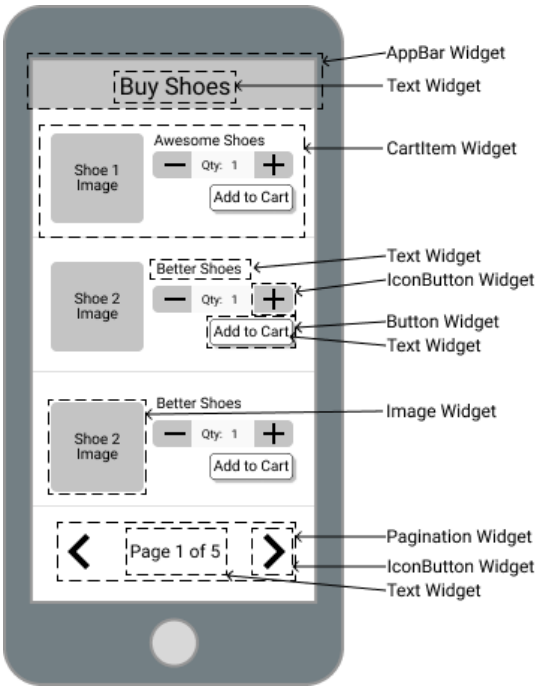


**Figure 1.8. Examples of structural widgets. There are far more of these, too, in reality.**




Thes are some of the most common widgets:

- Layout - `Row`, `Column`, `Scaffold`, `Stack`
- Structures - `Button`, `Toast`, `MenuDrawer` ,
- Styles - `TextStyle`, `Color`
- Animations - `FadeInPhoto`, tramsformations
- Positioning and Alignment - `Center`, `Padding`

### 1.7.2 Composing UI with Widgets

Flutter favors composition over class inheritance, which allows you to make your own unique wigets however you desire. All widgets are just a combination of smaller widgets.

In practice, that means that in Flutter you aren't sub-classing other widgets in order to build custom widget. Rather than do:

**Listing 1.1. This is wrong!**

```
class AddToCartButton extends Button {}
```

You would *compose* your button by wrapping the Button widget in other widgets. If you're coming from the web, this is similar to how React favors building small,

reusable components and combining them.

**Listing 1.2. Correct Flutter Widget Composition**

```
class AddToCartButton extends Widget {

  // ... class members
  @override
  build() {
    return Center(
      child: Button(
        child: Text('Add to Cart'),
      ),
    );
  }
}
```

Widgets have a few different overridable lifecycle methods and object members. The most important method, though, is `build()`. The build method *must* exist in each and every Flutter widget. This is the method in which you actually describe your view by returning Widgets.

### 1.7.3 Widget Types

There are (basically) two types of widgets: `Stateless` and `Stateful`. There are few other types of widgets, such as AnimatedWidgets, which we'll look at later. Those are actually just extensions of these two base classes, anyway.

A `StatelssWidget` is a widget which you (as the developer) are okay with being completely destroyed. In other words, there is no information kept within it that, if lost, would matter. All of its 'state' or data is passed into it. It's a 'dumb component', who's only jobs are to display informatin and look pretty. It's life depends on outside forces. It doesn't tell the framework when to remove it from the tree, or when to repaint it.

(As a side note, stateless widgets actually *are* reused by the framework in some situations, for performance, but thats all transparent to us. In practice, it's always destroyed completely.)

In our shopping cart, the `AddToCartButton` widget is stateless. It doesn't need to manage state, and it doesn't need to know about any other part of the tree. It's job is just to wait to be pressed, and then execute a function when that happens.

This doesn't mean that the Add to Cart button would never change. You might want to update it at some point to say 'Remove from Cart'. Some other widget would pass in the word to display, 'Add' or 'Remove', and it'll be repainted when the word being passed to it changes. It *reacts* to new information.

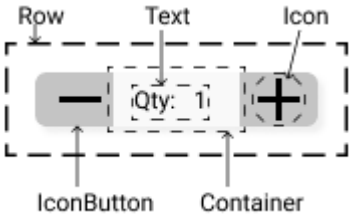
A `StatefulWidget` in the shopping cart application, on the other hand, would be the `QuantityCounter` widget, because it's managing the state that tracks the number of items you wish to add to your cart.

**NOTE** **A `StatefulWidget`, is a 'dumb widget', but it has an associated, smart `State` object. The state oject has special methods such as `setState` which tell Flutter when it needs to think about repainting.**

State objects are long-lived. They can tell Flutter to repaint, but can also be told to repaint becasue the associated stateful widget has been updated by outside forces. In this way, data can travel both ways.

Let's consider our shopping cart app again. So far, we know that on this screen there's quite a few components. These components are composed together via a combination of `Stateful` and `Stateless` widgets. Importantly, there is a `StatefulWidget` called `QuantityCounter`. This is a custom widget that I've created by by combining a variety of built in widgets.

**Figure 1.9. The quantity widget is composed of buttons, text fields, and layout widgets.**




The `build` object for this would look something like the following. I've pointed out the parts you should care about right now.

**Listing 1.3. Example of a build method for the custom QuantityCounter stateful widget.**

```
Widget build(BuildContext context) {
  return new Container(
    child: new Row(
      children: List<Widget>[
        new IconButton(
          icon: Icons.subtract,
          onPressed: () {
            setState(() {                    ❶
              this.quantity--;
            }
          }),
        new Text("Qty: ${this.quantity}),    ❷
        new IconButton(
          icon: Icons.add,
          onPressed: () {                    ❸
            setState(() {
              this.quantity++;
            }
          }),
      ],
    )
```

```
    );
}
```

1. Decrease the states quantity counter with `State.setState()`.
2. The widget will be repainted at this point in the tree every time the state object's quantity is called.
3. Increase the states quantity counter with `State.setState()`.

To be sure, there's a lot of layout and styling missing there, but that's the 'markup' you'd need.

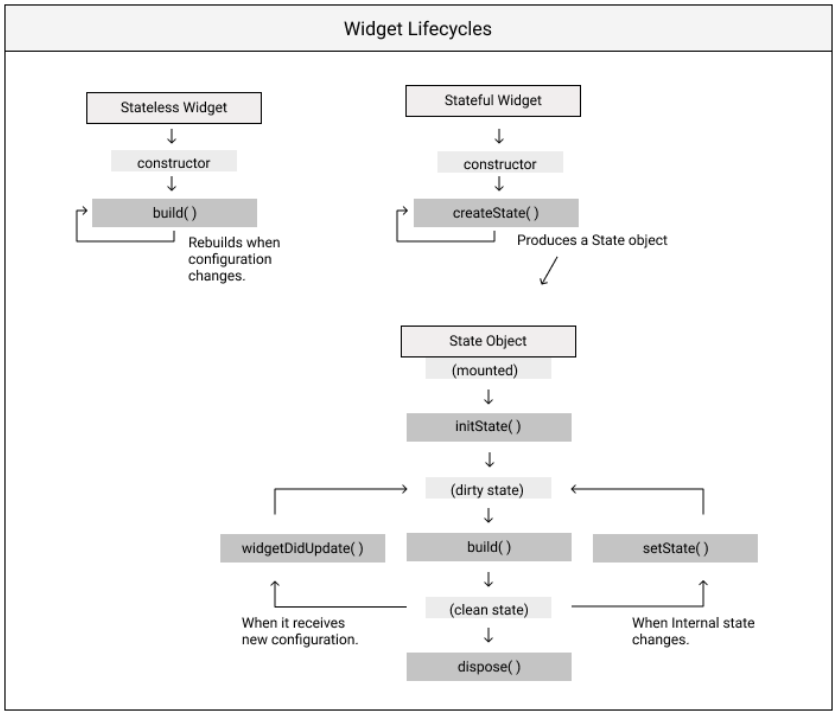
The important information in that code block, though, is what's going on inside the `IconButton`s and `Text` Widgets.

This widget has access to several methods from the base `StatefulWidget` class. The most important of which is `setState`. Everytime the "+" or "-" button is pressed, the app will call `setState`. This method will update whichever part of the widget's state you tell it to, and will tell Flutter to repaint the widgets that rely on the state change.

**Figure 1.10. Pressing the "+" button (a) increases the count, and repaints the text field (b)**


This process of building and updating widgets are part of it's 'lifecycle'. There are more proerties and methods on the state and stateful objects, which we'll explore in depth in later chapters.

**Figure 1.11. The two types of Flutter widgets have different lifecycles.**




Looking at the `QuantityWidget` again, it's lifecycle may look something like this:

1. When you navigate to the page, Flutter creates the class, which creates the `State` object associated with this widget.
2. As soon as the widget is 'mounted' Flutter calls `initState`.
3. After the state is initizialied, Flutter builds the widget (and as a consequence, renders it. You'll see that process in the next section.)
4. Now, the quantity widget is just sitting waiting for three possible actions:
   a. The use navigates to a different part of the app, in which case the `state` can be 'disposed'.
   b. Widgets outside of the this one in the tree have been updated and changed some sort of configuration that this widget relies on. In that case, this Widget's state calls `widgetDidUpdate`, and repaints if necessary. This may happen if the item sells out, and then a widget higher in the tree tells the this widget to 'disable' itself, since you could no longer add it to cart.
   c. Third, button is tapped, which calls `setState` and updates the widgets internal state. This also tells Flutter to rebuild and rerender.

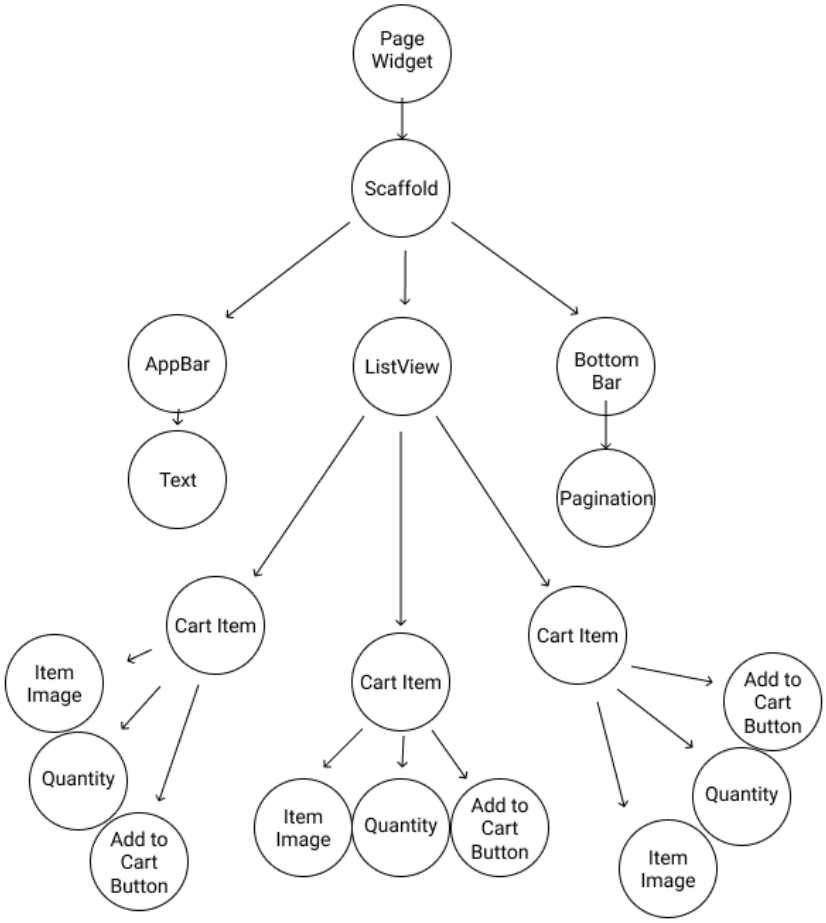
### *1.7.4 Flutter Rendering*

The special thing about the *developer expereince* in Flutter (paying no mind, at this moment, to how Flutter works under the hood), is the process that it does one million

times every day — the process by which Flutter builds (and rebuilds) your app.

At any moment, your Flutter app is composed of a giant widget tree. Here's a contrived example of what one page of the shopping cart's widget tree might look like. (In reality, it's much larger than this tree.) This tree is very similar to the DOM in web browsers.

**Figure 1.12. This represents a widget tree, but in reality there are far more widget in the tree then I can show.**

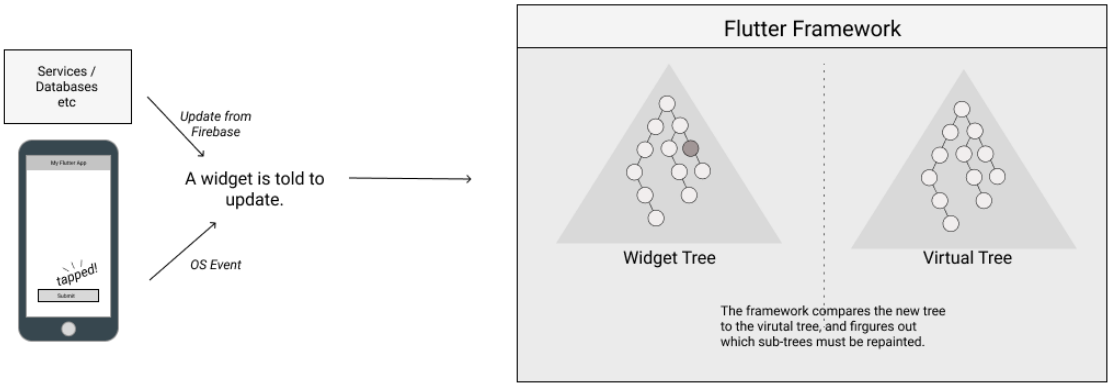



Take a look at the 'Quantity' widgets. Since they're are `Stateful`, all of their children likely rely on the state of that widget, and can likely delegate events to that widget to update the state. And update state starts the rendering process in sub-trees of your widget tree.

Flutter widgets are *reactive*. They respond to new information from an outside source, and Flutter rebuilds only what it needs to thanks to 'diffing', much like ReactJS. This is

an example of what happens when Flutter gets new information from user interaction.

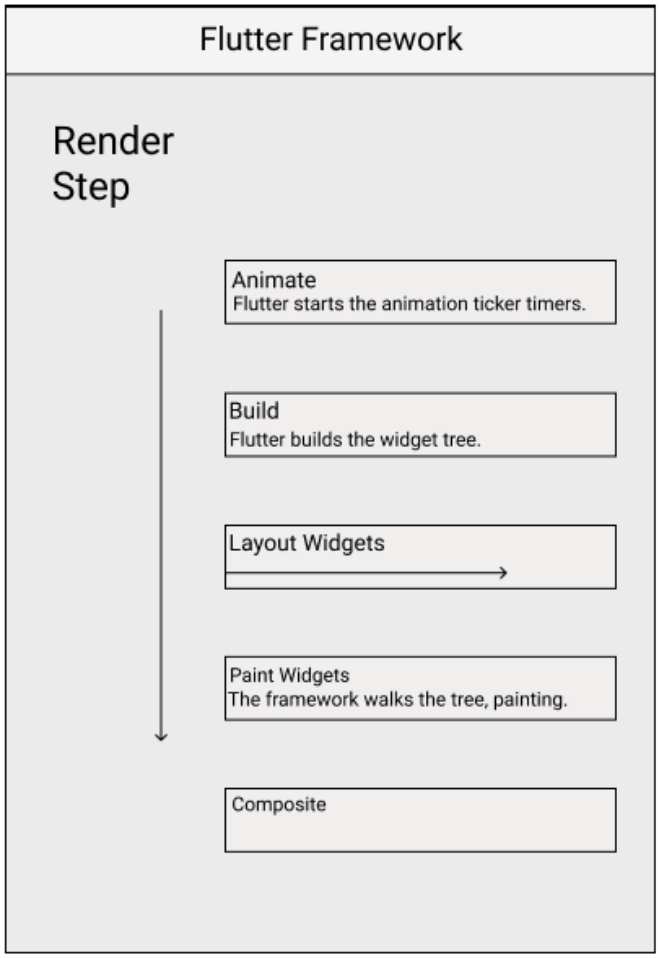
**Figure 1.13. The users taps a button, which changes some state, and the framework starts comparing the new tree to the previous tree, deciding which widgets need to be rebuilt. This is 'reactive'.**




1. A user taps that button.
2. The text widget is told to rebuild with it's new text.
3. Flutter compares its now changed widget tree to a copy of the previous widget tree. This is how the framework knows to only rebuild the parts of the tree that have changed, making it very efficient.

Now that Flutter has a new tree, it's ready to render. The render step is itself a series of steps.

**Figure 1.14. The high level steps of the rendering process.**




1. Flutter kicks off the render by starting any animation tickers. If you need to repaint, for example, because you're scrolling down a list, the starting position of any given element on the screen is incrementally moved to it's ending location, so that its a smooth animation. This is controlled by animation tickers, which dictate the *time* that an element has to move. This effectively controls how dramatic the animations are.

   Later in this book, we'll take a deep dive into animations in Flutter.

2. Next, Flutter builds all the widgets and the widget tree. By widgets, I mean the data that dictates the paint. When it 'builds' a button for the tree, it's not actually building a blue rectangle with text in it, that's a later step. It's building something like a tree node class:

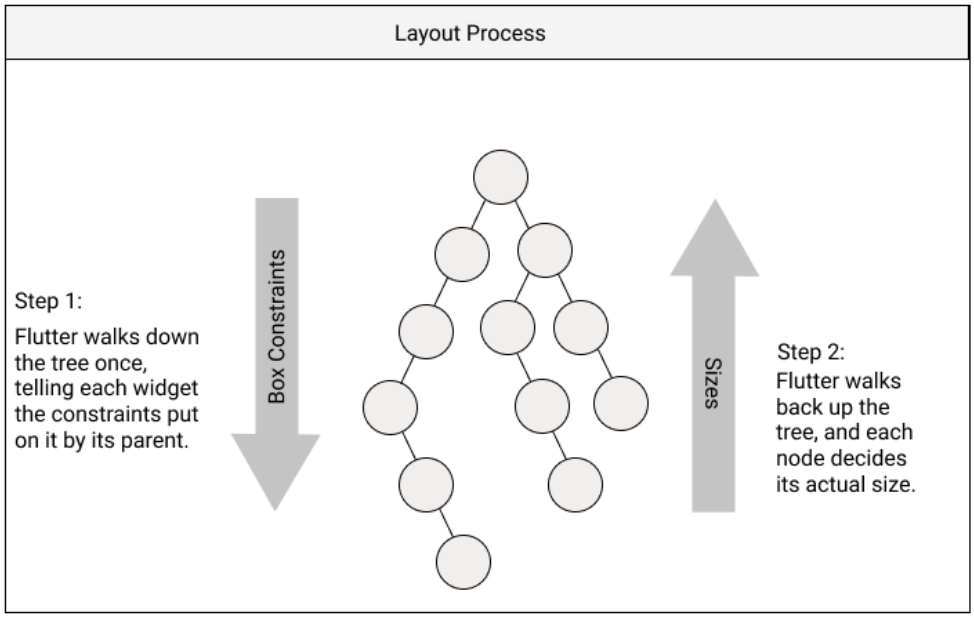
```
class Widget {
  child: another_tree_node
  color: red
  constraints: 120px x 140px
  etc
}
```

3. Now the tree is built, and ready to layout. Flutter walks down the tree, only once, in linear time. (If you aren't familiar with Big O notation, 'linear time' means 'fast'.) On the way down, it's collecting information about the size and position of widgets. In Flutter, layout and size constraints are dictated from parent to child, rather than the child itself knowing its size.

   On the way back up the tree, every widget now knows it's constraints, so the widget can tell Flutter it's *actual* size and position. The widgets are being laid out in relation to eachother.

   In the shopping cart example, this could mean that when the "" button is tapped on the `QuantityWidget`, and the state is updated with a new quantity, Flutter walks down the widget tree and `QuantityWidget` tells the buttons and text fields their _constraints_ (not actual size). Then the buttons will tell the widgets representing the "" and "-" icons *their* constraints, and so on down the tree. Once the algorithm hit's all the widgets at the very bottom, then all the widgets mut know their constraints, so on the way back up they can all safely take up the right amount of space and at the correct position.

**Figure 1.15. Flutter lays out all the widgets in one walk down and back up the tree, because widgets dictate their children's size constraints in Flutter.**

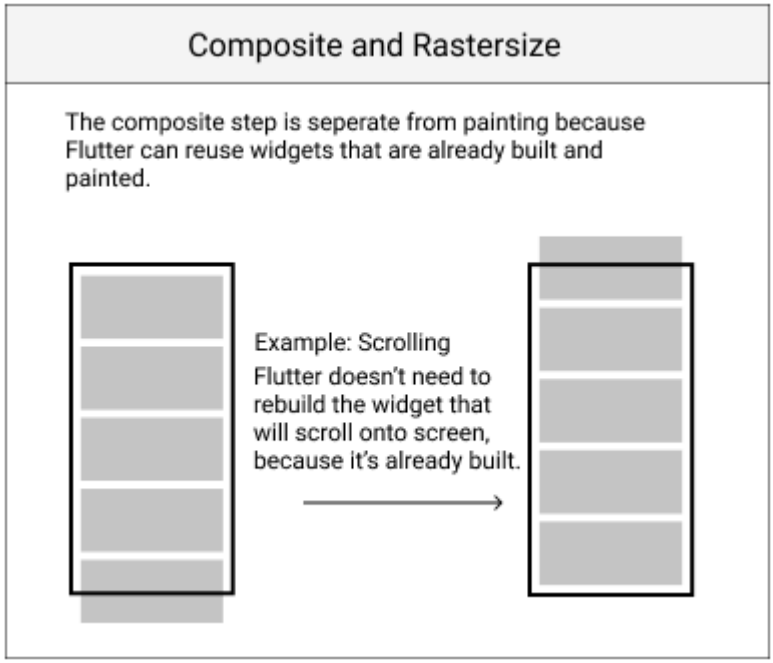


This single traveersal of the widget tree is incredibly powerful. By contrast, in a browser, layout is controlled by the DOM and CSS rules. Because of the 'cascading' nature of CSS and the fact that elements size and position can be controlled by their parents or themselves, it takes many walks down the 'DOM tree' to position all elements. This is one of the biggest performance bottlenecks on the modern web.

1. Now that each widget is laid out and know's it isn't conflicing with any other widgets, Flutter can paint the widgets. It's important to note here that the widgets still aren't 'rasterized', or physically paintd to pixels on the screen, that's coming up.
2. The composite step is next. The compositing step is when Flutter gives widgets their *actual* coordinates on the screen. They know the exact pixels they'll take up now.

   This step is purposefully and importantly seperate then the painting step. Because the widgets are prepainted, they can be reused before being painted again. This is useful, for example, when you're scrolling through a long list. Rather than rebuilding each of those list items every time a new one scrolls on or off the screen, Flutter already has them built and painted, and can just plug them in where they need to go.

**Figure 1.16. The composite step is separate than the painting step to increase perforamce.**




3. Finally, The widgets are ready to go. The engine combines the entire tree into a

renderable, single view, and tells the operating system to display it. This is called rasterizing.

We just covered an entire framework in a couple paragraphs. And it was a lot. You shouldn't be concerned yet if you don't know exactly how Flutter works, because I'll beat this dead horse throughout this entire book. These are the three main ideas that I want you to keep in mind as we dive deeper into Flutter, because these are the main ideas that matter to us as developers.

1. Flutter is reactive.
2. Everything is a widget.
3. Widget's size and contraints are dictated from their parents.

## 1.8 Final Note

If I can leave you with one impression at the end of this chapter, it's this: Flutter is simple to use and a powerful tool, but it does take effort to use well. It is, especially if you're a web-developer, a new paradigm of approaching UI. If you find yourself frustrated while learning from this book, don't assume it's your fault.

In fact, there are only two possible explanations here: First and most likely, you are just like every other human being, and programming is hard, and learning takes time. Re-read material, take a nap, and get back to it. You will get it.

The only other explanation is that I have done a poor job of making it digestable, and I welcome you to berate me on your choice of platform. I'm @ericwindmill everywhere.

## 1.9 Summary

- Flutter is mobile SDK written in Dart that empowers everyone to build beautiful and performant mobile apps.
- Dart is a language made by Google that can compile to JavaScript. It's fast, strictly typed, and easy to learn.
- The advantages of using Flutter are that it compiles to native device code, making it more performant than other cross-platform options. It also has the best developer experience around thanks to Darts JIT and Flutter's hot reload.
- Flutter is ideal for anyone that wants to make a highly performant cross-platform app fast. It's probably not the best choice for a large company with existing native teams.
- In Flutter, the *everything* is a **Widget**. Widgets are simply Dart classes that describe their view. UI is created by composing several small widgets into complete widget trees.
- Widgets come in two main flavors: `Stateless` and `Stateful`.
- Flutter provides state management tools, such as Widget lifecycle methods and special `State` objects.
- Flutter's rendering cycle is extremely fast because it's smart about what it needs to repaint.