



Open in app

Get started



Published in Dart



Kathy Walrath

Follow

Jul 25, 2019 · 5 min read · Listen



Save



# Dart asynchronous programming: Isolates and event loops

Dart, despite being a single-threaded language, offers support for futures, streams, background work, and all the other things you need to write in a modern, asynchronous, and (in the case of Flutter) reactive way. This article covers the foundations of Dart's support for background work: *isolates* and *event loops*.

If you prefer to learn by watching or listening, everything in this article is covered in the following video, which is part of the *Flutter in Focus* video series *Asynchronous Programming in Dart*:

## Isolates and Event Loops - Flutter in Focus





Open in app

Get started

## Isolates

An *isolate* is what all Dart code runs in. It's like a little space on the machine with its own, private chunk of memory and a single thread running an event loop.



An isolate has its own memory and a single thread of execution that runs an event loop.

In a lot of other languages like C++, you can have multiple threads sharing the same memory and running whatever code you want. In Dart, though, each thread is in its own isolate with its own memory, and the thread just processes events (more on that in a minute).

Many Dart apps run all their code in a single isolate, but you can have more than one if you need it. If you have a computation to perform that's so enormous it could cause you to drop frames if it were run in the main isolate, then you can use [`Isolate.spawn\(\)`](#) or [`Flutter's compute\(\) function`](#). Both of those functions create a separate isolate to do the number crunching, leaving your main isolate free to — say — rebuild and render the widget tree.





Open in app

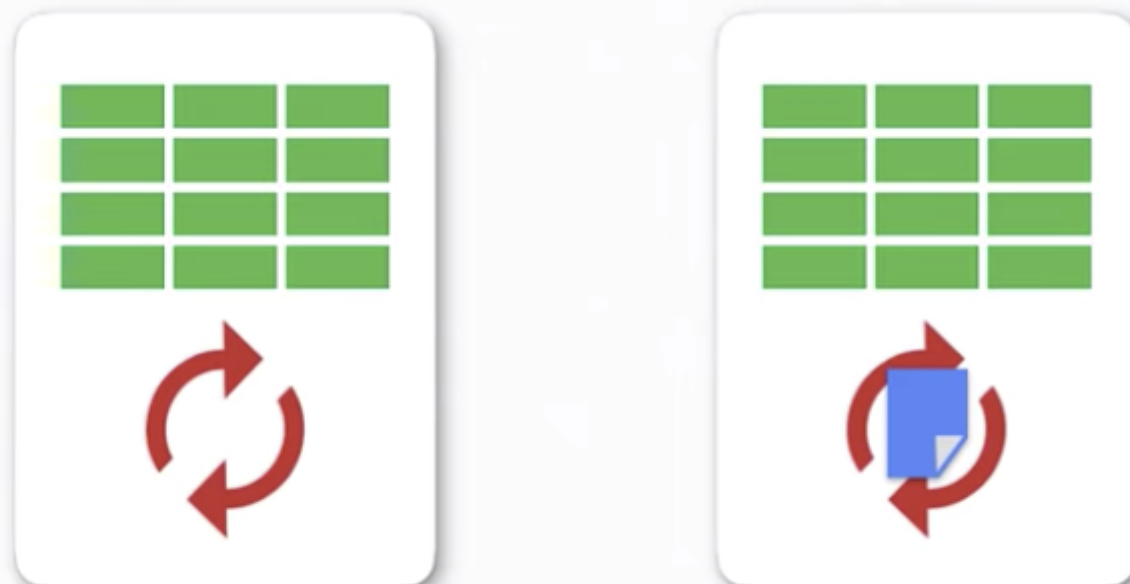
Get started



Two isolates, each with its own memory and thread of execution.

The new isolate gets its own event loop and its own memory, which the original isolate — even though it's the parent of this new one — isn't allowed to access. That's the source of the name *isolate*: these little spaces are kept *isolated* from one another.



In fact, the only way that isolates can work together is by passing messages back and forth. One isolate sends a message to the other, and the receiving isolate processes that message using its event loop.





Open in app

Get started

This lack of shared memory might  2.2K  7, especially if you're coming from a language like Java or C++, but it has some key benefits for Dart coders.

For example, memory allocation and garbage collection in an isolate don't require locking. There's only one thread, so if it's not busy, you know the memory isn't being mutated. That works out well for Flutter apps, which sometimes need to build up and tear down a bunch of widgets quickly.

## Event loops

Now that you've had a basic introduction to isolates, let's dig in to what really makes asynchronous code possible: the *event loop*.

Imagine the life of an app stretched out on a timeline. The app starts, the app stops, and in between a bunch of events occur — I/O from the disk, finger taps from the user... all kinds of stuff.

Your app can't predict when these events will happen or in what order, and it has to handle all of them with a single thread that never blocks. So, the app runs an event loop. It grabs the oldest event from its event queue, processes it, goes back for the next one, processes it, and so on, until the event queue is empty.

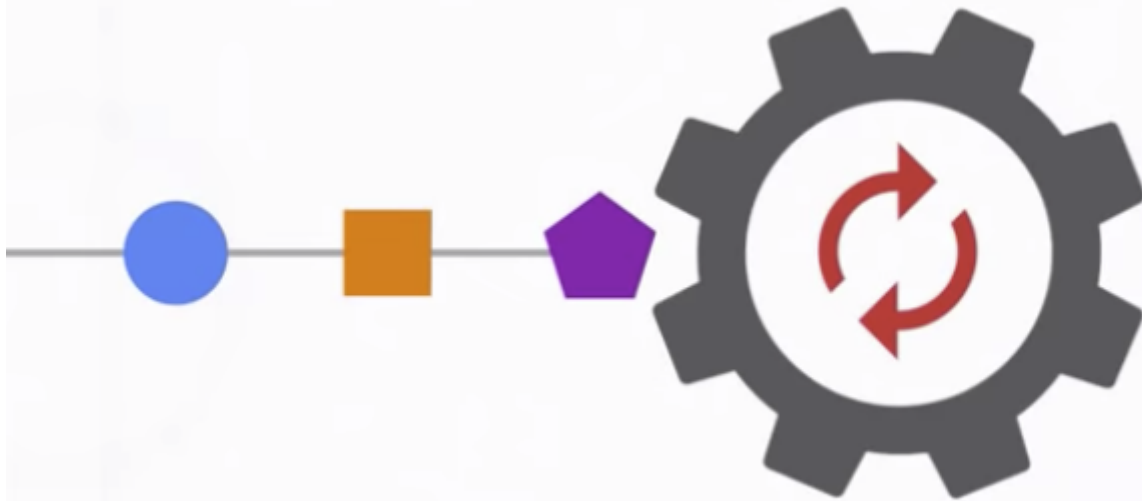
The whole time the app is running — you're tapping on the screen, things are downloading, a timer goes off — that event loop is just going around and around, processing those events one at a time.





Open in app

Get started



The event loop processes one event at a time.

When there's a break in the action, the thread just hangs out, waiting for the next event. It can trigger the garbage collector, get some coffee, whatever.

All of the high-level APIs and language features that Dart has for asynchronous programming — futures, streams, `async` and `await` — they're all built on and around this simple loop.

For example, say you have a button that initiates a network request, like this one:





Open in app

Get started

When you run your app, Flutter builds the button and puts it on screen. Then your app waits.

Your app's event loop just sort of idles, waiting for the next event. Events that aren't related to the button might come in and get handled, while the button sits there waiting for the user to tap on it. Eventually they do, and a tap event enters the queue.

That event gets picked up for processing. Flutter looks at it, and the rendering system says, "Those coordinates match the raised button," so Flutter executes the `onPressed` function. That code initiates a network request (which returns a future) and registers a completion handler for the future by using the `then()` method.

That's it. The loop is finished processing that tap event, and it's discarded.

Now, `onPressed` is a property of `RaisedButton`, and the network event uses a callback for a future, but both of those techniques are doing the same basic thing. They're both a way to tell Flutter, "Hey, later on, you might see a particular type of event come in. When you do, please execute this piece of code."

So, `onPressed` is waiting for a tap, and the future is waiting for network data, but from Dart's perspective, those are both just events in the queue.

And that's how asynchronous coding works in Dart. Futures, streams, `async` and `await` — these APIs are all just ways for you to tell Dart's event loop, "Here's some code, please run it later."



[Open in app](#)[Get started](#)

After you get used to working with asynchronous code, you'll start recognizing these patterns all over the place. Understanding the event loop is going to help as you move on to the higher level APIs.

## Summary

We took a quick look at isolates, the event loop, and the foundation of asynchronous



[Open in app](#)[Get started](#)

To learn more about asynchrony in Dart, check out the next article in this series, [Dart asynchronous programming: Futures](#).

*Big thanks to Andrew Brogdon, who created the video that this article is based on.*

Thanks to Andrew Brogdon and Deborah Owens

[About](#) [Help](#) [Terms](#) [Privacy](#)

Get the Medium app

