Creating streams in Dart

Written by Lasse Nielsen
April 2013 (updated May 2021)

The dart:async library contains two types that are important for many Dart APIs: Stream and Future. Where a Future represents the result of a single computation, a stream is a *sequence* of results. You listen on a stream to get notified of the results (both data and errors) and of the stream shutting down. You can also pause while listening or stop listening to the stream before it is complete.

But this article is not about *using* streams. It's about creating your own streams. You can create streams in a few ways:

- Transforming existing streams.
- Creating a stream from scratch by using an `async*` function.
- Creating a stream by using a `StreamController`.

This article shows the code for each approach and gives tips to help you implement your stream correctly.

For help on using streams, see Asynchronous Programming: Streams.

Transforming an existing stream

The common case for creating streams is that you already have a stream, and you want to create a new stream based on the original stream's events. For example you might have a stream of bytes that you want to convert to a stream of strings by UTF-8 decoding the input. The most general approach is to create a new stream that waits for events on the original stream and then outputs new events. Example:

```dart
/// Splits a stream of consecutive strings into lines.
///
/// The input string is provided in smaller chunks through
/// the `source` stream.
Stream<String> lines(Stream<String> source) async* {
  // Stores any partial line from the previous chunk.
  var partial = '';
  // Wait until a new chunk is available, then process it.
  await for (final chunk in source) {
    var lines = chunk.split('\n');
    lines[0] = partial + lines[0]; // Prepend partial line.
    partial = lines.removeLast(); // Remove new partial line.
    for (final line in lines) {
      yield line; // Add lines to output stream.
    }
  }
  // Add final partial line to output stream, if any.
  if (partial.isNotEmpty) yield partial;
}
```

For many common transformations, you can use `Stream`-supplied transforming methods such as `map()`, `where()`, `expand()`, and `take()`.

For example, assume you have a stream, `counterStream`, that emits an increasing counter every second. Here's how it might be implemented:

```dart
var counterStream =
    Stream<int>.periodic(const Duration(seconds: 1), (x) => x).take(15);
```

To quickly see the events, you can use code like this:

```dart
counterStream.forEach(print); // Print an integer every second, 15 times.
```

To transform the stream events, you can invoke a transforming method such as `map()` on the stream before listening to it. The method returns a new stream.

```dart
// Double the integer in each event.
var doubleCounterStream = counterStream.map((int x) => x * 2);
doubleCounterStream.forEach(print);
```

Instead of `map()`, you could use any other transforming method, such as the following:

```dart
.where((int x) => x.isEven) // Retain only even integer events.
.expand((var x) => [x, x]) // Duplicate each event.
.take(5) // Stop after the first five events.
```

Often, a transforming method is all you need. However, if you need even more control over the transformation, you can specify a StreamTransformer with `Stream`'s `transform()` method. The platform libraries provide stream transformers for many common tasks. For example, the following code uses the `utf8.decoder` and `LineSplitter` transformers provided by the dart:convert library.

```dart
Stream<List<int>> content = File('someFile.txt').openRead();
List<String> lines = await content
    .transform(utf8.decoder)
    .transform(const LineSplitter())
    .toList();
```

Creating a stream from scratch

One way to create a new stream is with an asynchronous generator (`async*`) function. The stream is created when the function is called, and the function's body starts running when the stream is listened to. When the function returns, the stream closes. Until the function returns, it can emit events on the stream by using `yield` or `yield*` statements.

Here's a primitive example that emits numbers at regular intervals:

```dart
Stream<int> timedCounter(Duration interval, [int? maxCount]) async* {
  int i = 0;
  while (true) {
    await Future.delayed(interval);
    yield i++;
    if (i == maxCount) break;
  }
}
```

This function returns a `Stream`. When that stream is listened to, the body starts running. It repeatedly delays for the requested interval and then yields the next number. If the `maxCount` parameter is omitted, there is no stop condition on the loop, so the stream outputs increasingly larger numbers forever - or until the listener cancels its subscription.

When the listener cancels (by invoking `cancel()` on the `StreamSubscription` object returned by the `listen()` method), then the next time the body reaches a `yield` statement, the `yield` instead acts as a `return` statement. Any enclosing `finally` block is executed, and the function exits. If the function attempts to yield a value before exiting, that fails and acts as a return.

When the function finally exits, the future returned by the `cancel()` method completes. If the function exits with an error, the future completes with that error; otherwise, it completes with `null`.

Another, more useful example is a function that converts a sequence of futures to a stream:

```dart
Stream<T> streamFromFutures<T>(Iterable<Future<T>> futures) async* {
  for (final future in futures) {
    var result = await future;
    yield result;
  }
}
```

This function asks the `futures` iterable for a new future, waits for that future, emits the resulting value, and then loops. If a future completes with an error, then the stream completes with that error.

It's rare to have an `async*` function building a stream from nothing. It needs to get its data from somewhere, and most often that somewhere is another stream. In some cases, like the sequence of futures above, the data comes from other asynchronous event sources. In many cases, however, an `async*` function is too simplistic to easily handle multiple data sources. That's where the `StreamController` class comes in.

Using a StreamController

If the events of your stream comes from different parts of your program, and not just from a stream or futures that can traversed by an `async` function, then use a StreamController to create and populate the stream.

A `StreamController` gives you a new stream and a way to add events to the stream at any point, and from anywhere. The stream has all the logic necessary to handle listeners and pausing. You return the stream and keep the controller to yourself.

The following example (from stream_controller_bad.dart) shows a basic, though flawed, usage of `StreamController` to implement the `timedCounter()` function from the previous examples. This code creates a stream to return, and then feeds data into it based on timer events, which are neither futures nor stream events.

bad

```dart
// NOTE: This implementation is FLAWED!
// It starts before it has subscribers, and it doesn't implement pause.
Stream<int> timedCounter(Duration interval, [int? maxCount]) {
  var controller = StreamController<int>();
  int counter = 0;
  void tick(Timer timer) {
    counter++;
    controller.add(counter); // Ask stream to send counter values as event.
    if (maxCount != null && counter >= maxCount) {
      timer.cancel();
      controller.close(); // Ask stream to shut down and tell listeners.
    }
  }

  Timer.periodic(interval, tick); // BAD: Starts before it has subscribers.
  return controller.stream;
}
```

As before, you can use the stream returned by `timedCounter()` like this:

```dart
var counterStream = timedCounter(const Duration(seconds: 1), 15);
counterStream.listen(print); // Print an integer every second, 15 times.
```

This implementation of `timedCounter()` has a couple of problems:

- It starts producing events before it has subscribers.
- It keeps producing events even if the subscriber requests a pause.

As the next sections show, you can fix both of these problems by specifying callbacks such as `onListen` and `onPause` when creating the `StreamController`.

Waiting for a subscription

As a rule, streams should wait for subscribers before starting their work. An `async*` function does this automatically, but when using a `StreamController`, you are in full control and can add events even when you shouldn't. When a stream has no subscriber, its `StreamController` buffers events, which can lead to a memory leak if the stream never gets a subscriber.

Try changing the code that uses the stream to the following:

```dart
void listenAfterDelay() async {
  var counterStream = timedCounter(const Duration(seconds: 1), 15);
  await Future.delayed(const Duration(seconds: 5));

  // After 5 seconds, add a listener.
  await for (final n in counterStream) {
    print(n); // Print an integer every second, 15 times.
  }
}
```

When this code runs, nothing is printed for the first 5 seconds, although the stream is doing work. Then the listener is added, and the first 5 or so events are printed all at once, since they were buffered by the `StreamController`.

To be notified of subscriptions, specify an `onListen` argument when you create the `StreamController`. The `onListen` callback is called when the stream gets its first subscriber. If you specify an `onCancel` callback, it's called when the controller loses its last subscriber. In the preceding example, `Timer.periodic()` should move to an `onListen` handler, as shown in the next section.

Honoring the pause state

Avoid producing events when the listener has requested a pause. An `async*` function automatically pauses at a `yield` statement while the stream subscription is paused. A `StreamController`, on the other hand, buffers events during the pause. If the code providing the events doesn't respect the pause, the size of the buffer can grow indefinitely. Also, if the listener stops listening soon after pausing, then the work spent creating the buffer is wasted.

To see what happens without pause support, try changing the code that uses the stream to the following:

```dart
void listenWithPause() {
  var counterStream = timedCounter(const Duration(seconds: 1), 15);
  late StreamSubscription<int> subscription;

  subscription = counterStream.listen((int counter) {
    print(counter); // Print an integer every second.
    if (counter == 5) {
      // After 5 ticks, pause for five seconds, then resume.
      subscription.pause(Future.delayed(const Duration(seconds: 5)));
    }
  });
}
```

When the five seconds of pause are up, the events fired during that time are all received at once. That happens because the stream's source doesn't honor pauses and keeps adding events to the stream. So the stream buffers the events, and it then empties its buffer when the stream becomes unpaused.

The following version of `timedCounter()` (from stream_controller.dart) implements pause by using the `onListen`, `onPause`, `onResume`, and `onCancel` callbacks on the `StreamController`.

```dart
Stream<int> timedCounter(Duration interval, [int? maxCount]) {
  late StreamController<int> controller;
  Timer? timer;
  int counter = 0;

  void tick(_) {
    counter++;
    controller.add(counter); // Ask stream to send counter values as event.
    if (counter == maxCount) {
      timer?.cancel();
      controller.close(); // Ask stream to shut down and tell listeners.
    }
  }

  void startTimer() {
    timer = Timer.periodic(interval, tick);
  }

  void stopTimer() {
    timer?.cancel();
    timer = null;
  }

  controller = StreamController<int>(
      onListen: startTimer,
      onPause: stopTimer,
      onResume: startTimer,
      onCancel: stopTimer);

  return controller.stream;
}
```

Run this code with the `listenWithPause()` function above. You'll see that it stops counting while paused, and it resumes nicely afterwards.

You must use all of the listeners—`onListen`, `onCancel`, `onPause`, and `onResume`—to be notified of changes in pause state. The reason is that if the subscription and pause states both change at the same time, only the `onListen` or `onCancel` callback is called.

Final hints

When creating a stream without using an async* function, keep these tips in mind:

- Be careful when using a synchronous controller—for example, one created using `StreamController(sync: true)`. When you send an event on an unpaused synchronous controller (for example, using the `add()`, `addError()`, or `close()` methods defined by EventSink), the event is sent immediately to all listeners on the stream. `Stream` listeners must never be called until the code that added the listener has fully returned, and using a synchronous controller at the wrong time can break this promise and cause good code to fail. Avoid using synchronous controllers.

- If you use `StreamController`, the `onListen` callback is called before the `listen` call returns the `StreamSubscription`. Don't let the `onListen` callback depend on the subscription already existing. For example, in the following code, an `onListen` event fires (and `handler` is called) before the `subscription` variable has a valid value.

  ```dart
  subscription = stream.listen(handler);
  ```

- The `onListen`, `onPause`, `onResume`, and `onCancel` callbacks defined by `StreamController` are called by the stream when the stream's listener state changes, but never during the firing of an event or during the call of another state change handler. In those cases, the state change callback is delayed until the previous callback is complete.

- Don't try to implement the `Stream` interface yourself. It's easy to get the interaction between events, callbacks, and adding and removing listeners subtly wrong. Always use an existing stream, possibly from a `StreamController`, to implement the `listen` call of a new stream.

- Although it's possible to create classes that extend `Stream` with more functionality by extending the `Stream` class and implementing the `listen` method and the extra functionality on top, that is generally not recommended because it introduces a new type that users have to consider. Instead of a class that *is* a `Stream` (and more), you can often make a class that *has* a `Stream` (and more).