Chapter 10. Flows

Up to now, we've covered coroutines, suspending functions, and how to deal with streams using `Channel`s. We've seen from the previous chapter that working with `Channel`s implies starting coroutines to send and/or receive from those `Channel`s. The aforementioned coroutines are then *hot* entities that are sometimes hard to debug, or can leak resources if they aren't cancelled when they should be.

`Flow`s, like `Channel`s, are meant to handle asynchronous streams of data, but at a higher level of abstraction and with better library tooling. Conceptually, `Flow`s are similar to `Sequence`s, except that each step of a `Flow` can be asynchronous. It is also easy to integrate flows in structured concurrency, to avoid leaking resources.

However, `Flow`s[1] aren't meant to replace `Channel`s. `Channel`s are building blocks for flows. `Channel`s are still appropriate in some architectures such as in CSP (see Chapter 9). Nevertheless, you'll see that flows suit most needs in asynchronous data processing.

In this chapter, we'll introduce you to cold and hot flows. You'll see how *cold* flows can be a better choice when you want to make sure never to leak any resources. On the other hand, *hot* flows serve a different purpose such as when you need a "publish-subscribe" relationship between entities in your app. For example, you can implement an event bus using hot flows.

The best way to understand flows is to see how they are used in real-life applications. So this chapter will also go through a series of typical use cases.

An Introduction to Flows

Lets reimplement Example 9-6, using a `Flow`:

```
fun numbers(): Flow<Int> = flow {
    emit(1)
    emit(2)
    // emit other values
}
```

Several aspects are important to notice:

1. Instead of returning a `Channel` instance, we're returning a `Flow` instance.
2. Inside the flow, we use the `emit` suspending function instead of `send`.
3. The `numbers` function, which returns a `Flow` instance, isn't a suspending function. Invoking the `numbers` function doesn't start anything by itself—it just immediately returns a `Flow` instance.

To sum up, you define in the `flow` block the emission of values. When invoked, the `numbers` function quickly returns a `Flow` instance without running anything in the background.

On the consuming site:

```
fun main() = runBlocking {
    val flow = numbers()      ❶
    flow.collect {            ❷
        println(it)
    }
}
```

❶ We get an instance of `Flow`, using the `numbers` function.

❷ Once we get a flow, instead of looping over it (like we would with a channel), we use the `collect` function which, in flows parlance, is called a *terminal operator*. We'll extend on *flows operators* and terminal operators in "Operators". For now, we can summarize the purpose of the `collect` terminal operator: it consumes the flow; foor example, iterate over the flow and execute the given lambda on each element of the flow.

That's it—you've seen the basic usage of a flow. As we mentioned earlier, we'll now take a more realistic example, so you'll see the real interest of `Flow`s.

A More Realistic Example

Imagine that you need to get tokens from a remote database,[2] then query additional data for each of those tokens. You need to do that only once in a while, so you decide not to maintain an active connection to the database (which could be expensive). So you create a connection only when fetching the data, and close it when you're done.

Your implementation should first establish the connection to the database. Then you get a token using a suspending function `getToken`. This `getToken` function performs a request to the database and returns a token. Then you asynchronously get optional data associated with this token. In our example, this is done by invoking the suspending function `getData`, which takes a token as a parameter. Once you get the result of `getData`, you wrap both the token and the result in one `TokenData` class instance, defined as:

```
data class TokenData(val token: String, val opt: String? = null)
```

To sum up, you need to produce a stream of `TokenData` objects. This stream requires first establishing a database connection, then performing asynchronous queries for retrieving tokens and getting associated data. You choose how many tokens you need. After you've processed all the tokens, you disconnect and release underlying database connection resources. Figure 10-1 shows how to implement such a flow.

Figure 10-1. Data flow.

You can find the corresponding source code in GitHub.

NOTE

In this chapter, we sometimes use images instead of code blocks because the screenshots from our IDE show suspension points (in the margin) and type hints, which are really helpful.

Several aspects of this implementation are particularly important to notice:

- Creating a connection to the database and closing it on completion is completely transparent to the client code that consumes the flow. Client code only sees a flow of `TokenData`.
- All operations inside the flow are sequential. For example, once we get the first token (say, "token1"), the flow invokes `getData("token1")` and suspends until it gets the result (say, "data1"). Then the flow emits the first `TokenData("token1,"` `"data1")`. Only after that does the execution proceed with "token2," etc.
- Invoking the `getDataFlow` function does nothing on its own. It simply returns a flow. The code inside the flow executes only when a coroutine collects the flow, as shown in Example 10-1.

  Example 10-1. Collecting a flow

  ```
  fun main() = runBlocking<Unit> {
      val flow = getDataFlow(3) // Nothing runs at initialization

      // A coroutine collects the flow
      launch {
          flow.collect { data ->
              println(data)
          }
      }
  }
  ```

- If the coroutine that collects the flow gets cancelled or reaches the end of the flow, the code inside the `onCompletion` block executes. This guarantees that we properly release the connection to the database.

As we already mentioned, `collect` is a terminal operator that consumes all elements of the flow. In this example, `collect` invokes a function on each collected element of the flow (e.g., `println(data)` is invoked three times). We'll cover other terminal operators in "Examples of Cold Flow Usage".

NOTE

Until now, you've seen examples of flows that don't run any code until a coroutine collects them. In flows parlance, they are cold flows.

Operators

If you need to perform transformations on a flow, much like you would do on collections, the coroutines library provides functions such as `map`, `filter`, `debounce`, `buffer`, `onCompletion`, etc. Those functions are called *flow operators* or *intermediate operators*, because they operate on a

flow and return another flow. A regular operator shouldn't be confused with a terminal operator, as you'll see later.

In the following, we have an example usage of the `map` operator:

```
fun main() = runBlocking<Unit> {
    val numbers: Flow<Int> = // implementation hidden for brevity

    val newFlow: Flow<String> = numbers().map {
        transform(it)
    }
}

suspend fun transform(i :Int): String = withContext(Dispatchers.Default) {
    delay(10) // simulate real work
    "${i + 1}"
}
```

The interesting bit here is that `map` turns a `Flow<Int>` into a `Flow<String>`. The type of the resulting flow is determined by the return type of the lambda passed to the operator.

NOTE

The `map` flow operator is conceptually really close to the `map` extension function on collections. There's a noticeable difference, though: the lambda passed to the `map` flow operator can be a suspending function.

We'll cover most of the common operators in a series of use cases in the next section.

Terminal Operators

A terminal operator can be easily distinguished from other regular operators since it's a suspending function that starts the collection of the flow. You've previously seen `collect`.

Other terminal operators are available, like `toList`, `collectLatest`, `first`, etc. Here is a brief description of those terminal operators:

- `toList` collects the given flow and returns a `List` containing all collected elements.
- `collectLatest` collects the given flow with a provided action. The difference from `collect` is that when the original flow emits a new value, the action block for the previous value is cancelled.
- `first` returns the first element emitted by the flow and then cancels the flow's collection. It throws a `NoSuchElementException` if the flow was empty. There's also a variant, `firstOrNull`, which returns `null` if the flow was empty.

Examples of Cold Flow Usage

As it turns out, picking one single example making use of all possible operators isn't the best path to follow. Instead, we'll provide different use cases, which will illustrate the usage of several flow operators.

Use Case #1: Interface with a Callback-Based API

Suppose that you're developing a chat application. Your users can send messages to one another. A message has a date, a reference to the author of the message, and content as plain text.

Here is a `Message`:

```
data class Message(
    val user: String,
    val date: LocalDateTime,
    val content: String
)
```

Unsurprisingly, we'll represent the stream of messages as a flow of the `Message` instance. Every time a user posts a message into the app, the flow will transmit that message. For now, assume that you can invoke a function `getMessageFlow`, which returns an instance of `Flow<Message>`. With the Kotlin Flows library, you are able to create your own custom flows. However, it makes the most sense to start by exploring how the flow API can be used in common use cases:

```
fun getMessageFlow(): Flow<Message> {
    // we'll implement it later
}
```

Now, suppose that you want to translate all messages from a given user in a different language, on the fly. Moreover, you'd like to perform the translation on a background thread.

To do that, you start by getting the flow of messages, by invoking `getMessageFlow()`. Then you apply operators to the original flow, as shown in the following:

```
fun getMessagesFromUser(user: String, language: String): Flow<Message> {
    return getMessageFlow()
        .filter { it.user == user }           ❶
        .map { it.translate(language) }       ❷
        .flowOn(Dispatchers.Default)          ❸
}
```

❶ The first operator, `filter`, operates on the original flow and returns another flow of messages which all originate from the same `user` passed as a parameter.

❷ The second operator, `map`, operates on the flow returned by `filter` and returns a flow of translated messages. From the `filter` operator standpoint, the original flow (returned by `getMessageFlow()`) is the *upstream flow*, while the *downstream flow* is represented by all operators happening *after* `filter`. The same reasoning applies for all intermediate operators—they have their own relative upstream and downstream flow, as illustrated in Figure 10-2.

❸ Finally, the `flowOn` operator changes the context of the flow it is operating on. It changes the coroutine context of the upstream

flow, while not affecting the downstream flow. Consequently, steps 1 and 2 are done using the dispatcher `Dispatchers.Default`.

In other words, the upstream flow's operators (which are `filter` and `map`) are now encapsulated: their execution context will always be `Dispatchers.Default`. It doesn't matter in which context the resulting flow will be collected; the previously mentioned operators will be executed using `Dispatchers.Default`.

This is a very important property of flows, called *context preservation.* Imagine that you're collecting the flow on the UI thread of your application—typically, you would do that using the `viewModelScope` of a `ViewModel`. It would be embarrassing if the context of execution of one of the flow's operators leaked downstream and affected the thread in which the flow was ultimately collected. Thankfully, this will never happen. For example, if you collect a flow on the UI thread, all values are emitted by a coroutine that uses `Dispatchers.Main`. All the necessary context switches are automatically managed for you.

Figure 10-2. Upstream and downstream flows.

Under the hood, `flowOn` starts a new coroutine when it detects that the context is about to change. This new coroutine interacts with the rest of the flow through a channel that is internally managed.

NOTE

In flow parlance, an intermediate operator like `map` operates on the upstream flow and returns another flow. From the `map` operator standpoint, the returned flow is the downstream flow.

The `map` operator accepts a suspending function as a transformation block. So if you wanted to only perform message translation using `Dispatchers.Default` (and not message filtering), you could remove the `flowOn` operator and declare the `translate` function like so:

```
private suspend fun Message.translate(
    language: String
): Message  = withContext(Dispatchers.Default) {
    // this is a dummy implementation
    copy(content = "translated content")
}
```

See how easy it is to offload parts of data transformation to other threads, while still having a big picture of the data flow?

As you can see, the Flow API allows for a declarative way to express data transformation. When you invoke `getMessagesFromUser("Amanda," "en-us")`, nothing is actually running. All those transformations involve intermediate operators, which will be triggered when the flow will be collected.

On the consuming site, if you need to act on each received message, you can use the `collect` function like so:

```
fun main() = runBlocking {
    getMessagesFromUser("Amanda", "en-us").collect {
        println("Received message from ${it.user}: ${it.content}")
    }
}
```

Now that we've shown how to transform the flow and consume it, we can provide an implementation for the flow itself: the `getMessageFlow` function. The signature of this function is to return a flow of `Message`s. In that particular situation, we can reasonably assume that the message machinery is actually a service that runs in its own thread. We'll name this service `MessageFactory`.

Like most services of that kind, the message factory has a *publish/subscribe* mechanism—we can register or unregister observers for new incoming messages, as shown in the following:

```
abstract class MessageFactory : Thread() {
    /* The internal list of observers must be thread-safe */
    private val observers = Collections.synchronizedList(
        mutableListOf<MessageObserver>())
    private var isActive = true

    override fun run() = runBlocking {
        while(isActive) {
            val message = fetchMessage()
            for (observer in observers) {
                observer.onMessage(message)
            }
```

```
                delay(1000)
            }
        }

        abstract fun fetchMessage(): Message

        fun registerObserver(observer: MessageObserver) {
            observers.add(observer)
        }

        fun unregisterObserver(observer: MessageObserver) {
            observers.removeAll { it == observer }
        }

        fun cancel() {
            isActive = false
            observers.forEach {
                it.onCancelled()
            }
            observers.clear()
        }

        interface MessageObserver {
            fun onMessage(msg: Message)
            fun onCancelled()
            fun onError(cause: Throwable)
        }
    }
```

This implementation polls for new messages every second and notifies observers. Now the question is: how do we turn a hot[3] entity such as this `MessageFactory` into a flow? `MessageFactory` is also said to be *callback-based*, because it holds references to `MessageObserver` instances and calls methods on those instances when new messages are retrieved. To bridge the flow world with the "callback" world, you can use the `callbackFlow` flow builder. Example 10-2 shows how you can use it.

Example 10-2. Making a flow from a callback-based API

```
fun getMessageFlow(factory: MessageFactory) = callbackFlow<Message> {
    val observer = object : MessageFactory.MessageObserver {
        override fun onMessage(msg: Message) {
            trySend(msg)
        }

        override fun onCancelled() {
            channel.close()
        }

        override fun onError(cause: Throwable) {
            cancel(CancellationException("Message factory error", cause))
        }
    }

    factory.registerObserver(observer)
    awaitClose {
        factory.unregisterObserver(observer)
    }
}
```

The `callbackFlow` builder creates a cold flow which doesn't perform anything until you invoke a terminal operator. Let's break it down. First

off, it's a parameterized function which returns a `Flow` of the given type. It's always done in three steps:

```
callbackFlow {
    /*
    1. Instantiate the "callback." In this case, it's an observer.
    2. Register that callback using the available api.
    3. Listen for close event using `awaitClose`, and provide a
       relevant action to take in this case. Most probably,
       you'll have to unregister the callback.
    */
}
```

It's worth having a look at the signature of `callbackFlow`:

```
public inline fun <T> callbackFlow(
    @BuilderInference noinline block: suspend ProducerScope<T>.() -> Unit
): Flow<T>
```

Don't be impressed by this. One key piece of information is that `callbackFlow` takes a suspending function with `ProducerScope` receiver as the argument. This means that inside the curly braces of the block following `callbackFlow`, you have a `ProducerScope` instance as an implicit `this`.

Here is the signature of `ProducerScope`:

```
public interface ProducerScope<in E> : CoroutineScope, SendChannel<E>
```

So a `ProducerScope` is a `SendChannel`. And that's what you should remember: `callbackFlow` provides you with an instance of `SendChannel`, which you can use inside your implementation. You send the object instances you get from your callback to this channel. This is what we do in step 1 of Example 10-2.

Use Case #2: Concurrently Transform a Stream of Values

Sometimes you have to apply a transformation on a collection or stream of objects, to get a new collection of transformed objects. When those transformations should be done asynchronously, things start getting a bit complicated. Not with flows!

Imagine that you have a list of `Location` instances. Each location can be resolved to a `Content` instance, using the `transform` function:

```
suspend fun transform(loc: Location): Content = withContext(Dispatchers.IO) {
    // Actual implementation doesn't matter
}
```

So you are receiving `Location` instances, and you have to transform them on the fly using the `transform` function. However, processing one `Location` instance might take quite some time. So you don't want that processing of a location to delay the transformation of the next incoming

locations. In other words, transformations should be done *in parallel*, as shown in Figure 10-3.

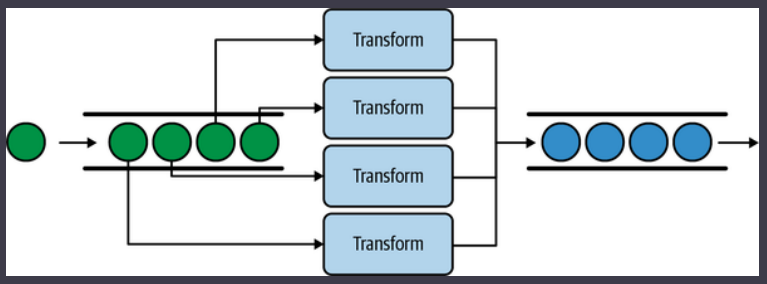

Figure 10-3. Merge flows.

In the preceding schema, we've limited the concurrency to four; in other words, at most, four locations can be transformed simultaneously at a given point in time.

Figure 10-4 shows how you would implement this behavior using flows.

Figure 10-4. Implementing merging flows.

You can find the corresponding source code in GitHub.

To understand what's going on here, you should realize that `locations.map{..}` returns a flow of a flow (e.g., the type is `Flow<Flow<Content>>`). Indeed, inside the `map{..}` operator, a new flow is created upon emission of a location by the upstream flow (which is `locationsFlow`). Each of those created flows is of type `Flow<Content>` and individually performs location transformation.

The last statement, `flattenMerge`, merges all those created flows inside a new resulting `Flow<Content>` (which we assign to `contentFlow`). Also, `flattenMerge` has a "concurrency" parameter. Indeed, it would probably be inappropriate to concurrently create and collect a flow every time we receive a location. With a concurrency level of 4, we ensure that no more than four flows will be collected at a given point in time. This is handy in the case of CPU-bound tasks, when you know that your CPU won't be able to transform more than four locations *in parallel* (assuming the CPU has four cores). In other words, `flattenMerge`'s concurrency level refers to how many operations/transformations will be done in parallel *at most* at a given point in time.

Thanks to the suspending nature of flows, you get *back pressure* for free. New locations are collected from `locationsFlow` only when the machinery is available to process them. A similar mechanism could be implemented without flows or coroutines, using a thread pool and a blocking queue. However, that would require considerably more lines of code.

NOTE

As of this writing, the `flattenMerge` operator is marked as `@FlowPreview` in the source code, which means that this declaration is in a preview state and can be changed in a backward-incompatible manner with a best-effort migration.

We hope that by the time we finish writing this book, the flow-merging API will be stabilized. Otherwise, a similar operator might replace `flattenMerge`.

What Happens in Case of Error?

If one of the `transform` functions raises an exception, the entire flow will be cancelled, and the exception will be propagated downstream. While this good default behavior, you might want to handle some exceptions right inside the flow itself.

We'll show how to do that in "Error Handling".

Final Thoughts

- Do you realize that we've just created a worker pool that concurrently transforms an incoming stream of objects, using only five lines of code?
- You're guaranteed that the flow machinery is thread-safe. No more headaches figuring out the proper synchronization strategy to pass object references from a thread pool to a collecting thread.
- You can easily tweak the concurrency level, which, in this case, means the maximum number of parallel transformations.

Use Case #3: Create a Custom Operator

Even if a lot of flow operators are available out of the box, sometimes you'll have to make your own. Thankfully, flows are composable, and it's not that difficult to implement custom reactive logic.

For example, by the time we write those lines, there's no Flows operator equivalent of the Project Reactor's bufferTimeout.

So, what is `bufferTimeout` supposed to do? Imagine that you have an upstream flow of elements, but you want to process those elements by batches and at a fixed maximum rate. The flow returned by `bufferTimeout` should buffer elements and emit a list (batch) of elements when either:

- The buffer is full.
- A predefined maximum amount of time has elapsed (timeout).

Before going through the implementation, let's talk about the key idea. The flow returned by `bufferTimeout` should internally consume the upstream flow and buffer elements. When the buffer is full, or a timeout has elapsed, the flow should emit the content of the buffer (a list). You can imagine that internally we'll start a coroutine that receives two types of events:

- "An element has just been received from the upstream flow. Should we just add it to the buffer or also send the whole buffer?"
- "Timeout! Send the content of the buffer right now."

In Chapter 9 (CSP section), we've discussed a similar situation. The `select` expression is perfect for dealing with multiple events coming from several channels.

Now we're going to implement our `bufferTimeout` flow operator:

You can find the corresponding source code in GitHub.

Here is the explanation:

- First of all, the signature of the operator tells us a lot. It's declared as an extension function of `Flow<T>`, so you can use it like this: `upstreamFlow.bufferTimeout(10, 100)`. As for the return type, it's `Flow<List<T>>`. Remember that you want to process elements by batches, so the flow returned by `bufferTimeout` should return elements as `List<T>`.
- Line 17: we're using a `flow{}` builder. As a reminder, the builder provides you an instance of `FlowCollector`, and the block of code is an extension function with `FlowCollector` as the receiver type. In other words, you can invoke `emit` from inside the block of code.
- Line 21: we're using `coroutineScope{}` because we'll start new coroutines, which is only possible within a `CoroutineScope`.
- Line 22: from our coroutine standpoint,[4] received elements should come from a `ReceiveChannel`. So another inner coroutine should be started to consume the upstream flow and send them over a channel. This is exactly the purpose of the `produceIn` flow operator.
- Line 23: we need to generate "timeout" events. A library function already exists exactly for that purpose: `ticker`. It creates a channel that produces the first item after the given initial delay, and subsequent items with the given delay between them. As specified in the documentation, `ticker` starts a new coroutine *eagerly*, and we're fully responsible for cancelling it.
- Line 34: we're using `whileSelect`, which really is just syntax sugar for looping in a `select` expression while clauses return `true`. Inside the `whileSelect{}` block you can see the logic of adding an element to the buffer only if it's not full, and emitting the whole buffer otherwise.
- Line 46: when the upstream flow collection completes, the coroutine started with `produceIn` will still attempt to read from that flow, and a `ClosedReceiveChannelException` will be raised. So we catch that exception, and we know that we should emit the content of the buffer.
- Lines 48 and 49: channels are hot entities—they should be cancelled when they're not supposed to be used anymore. As for the `ticker`, it should be cancelled too.

Usage

Figure 10-5 shows an example of how `bufferTimeout` can be used.

Figure 10-5. `bufferTimeout` usage.

You can find the corresponding source code in GitHub.

The output is:

```
139 ms: [1, 2, 3, 4]
172 ms: [5, 6, 7, 8]
223 ms: [9, 10, 11, 12, 13]
272 ms: [14, 15, 16, 17]
322 ms: [18, 19, 20, 21, 22]
...
```

```
1022 ms: [86, 87, 88, 89, 90]
1072 ms: [91, 92, 93, 94, 95]
1117 ms: [96, 97, 98, 99, 100]
```

As you can see, the upstream flow is emitting numbers from 1 to 100, with a delay of 10 ms between each emission. We set a timeout of 50 ms, and each emitted list can contain at most five numbers.

Error Handling

Error handling is fundamental in reactive programming. If you're familiar with RxJava, you probably handle exceptions using the `onError` callback of the `subscribe` method:

```
// RxJava sample
someObservable().subscribe(
    { value -> /* Do something useful */ },
    { error -> println("Error: $error") }
)
```

Using flows, you can handle errors using a combination of techniques, involving:

- The classic `try`/`catch` block.
- The `catch` operator—we'll cover this new operator right after we discuss the `try`/`catch` block.

The try/catch Block

If we define a dummy upstream flow made of only three `Int`s, and purposely throw an exception inside the `collect{}` block, we can catch the exception by wrapping the whole chain in a `try`/`catch` block:

You can find the corresponding source code in GitHub.

The output is:

```
Received 1
Received 2
Caught java.lang.RuntimeException
```

It is important to note that `try`/`catch` also works when the exception is raised from inside the upstream flow. For example, we get the exact same result if we change the definition of the upstream flow to:

You can find the corresponding source code in GitHub.

However, if you try to intercept an exception in the flow itself, you're likely to get unexpected results. Here is an example:

```
// Warning: DON'T DO THIS, this flow swallows downstream exceptions
val upstream: Flow<Int> = flow {
    for (i in 1..3) {
        try {
            emit(i)
        } catch (e: Throwable) {
            println("Intercept downstream exception $e")
        }
    }
}

fun main() = runBlocking {
    try {
        upstream.collect { value ->
            println("Received $value")
            check(value <= 2) {
                "Collected $value while we expect values below 2"
            }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

In this example, we're using the `flow` builder to define `upstream`, and we wrapped the `emit` invocation inside a `try`/`catch` statement. Even if it seems useless because `emit` isn't throwing exceptions, it could make sense with nontrivial emission logic nevertheless. At the consuming site, in the `main` function, we collect that flow and we check that we don't get values strictly greater than 2. Otherwise, the `catch` block should print `Caught java.lang.IllegalStateException Collected x while we expect values below 2`.

We expect the following output:

```
Received 1
Received 2
Caught java.lang.IllegalStateException: Collected 3 whil
```

However, this is what we actually get:

```
Received 1
Received 2
Received 3
Intercept downstream exception java.lang.IllegalStateExc
```

Despite the exception raised by `check(value <= 2) {..}`, that exception gets caught not by the `try`/`catch` statement of the `main` function, but by the `try`/`catch` statement of the flow.

WARNING

A `try`/`catch` statement inside a flow builder might catch *downstream* exceptions—which includes exceptions raised during the collection of the flow.

Separation of Concern Is Important

A flow implementation shouldn't have a side effect on the code that collects that flow. Likewise, the code that collects a flow shouldn't be aware of the implementation details of the upstream flow. A flow should always be *transparent to exceptions*: it should propagate exceptions coming from a collector. In other words, a flow should never swallow downstream exceptions.

Throughout this book, we'll refer to *exception transparency* to designate a flow that is *transparent to exceptions*.

Exception Transparency Violation

The previous example was an example of exception transparency violation. Trying to emit values from inside a `try`/`catch` block is another violation. Here is an example (again, don't do this!):

```
val violatesExceptionTransparency: Flow<Int> = flow {
    for (i in 1..3) {
        try {
            emit(i)
        } catch (e: Throwable) {
            emit(-1)
        }
    }
}

fun main() = runBlocking {
    try {
        violatesExceptionTransparency.collect { value ->
            check(value <= 2) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

The output is:

```
Caught java.lang.IllegalStateException: Flow exception t
violated:
Previous 'emit' call has thrown exception java.lang.Ille
Emissions from 'catch' blocks are prohibited in order to
For a more detailed explanation, please refer to Flow do
```

The `try`/`catch` block should *only* be used to surround the collector, to handle exceptions raised from the collector itself, or (possibly, although it's not ideal) to handle exceptions raised from the flow.

To handle exceptions inside the flow, you should use the `catch` operator.

The catch Operator

The `catch` operator allows for a declarative style of catching exceptions, as shown in Figure 10-6. It catches all upstream exceptions. By all exceptions, we mean that it even catches `Throwable`s. Since it only catches upstream exceptions, the `catch` operator doesn't have the exception issue of the `try`/`catch` block.

Figure 10-6. Declarative style.

You can find the corresponding source code in GitHub.

The output is:

```
Received 1
Received 2
Caught java.lang.RuntimeException
```

The flow raises a `RuntimeException` if it's passed a value greater than 2. Right after, in the `catch` operator, we print in the console. However, the collector never get the value 3. So the `catch` operator automatically cancels the flow.

Exception transparency

From inside this operator, you can only catch *upstream exceptions*. When we say upstream, we mean relative to the `catch` operator. To show what we mean, we'll pick an example where the collector throws an exception before the flow internally throws another exception. The collector should be able to catch the raised exception (the exception shouldn't be caught by the flow):

You can find the corresponding source code in GitHub.

In this example, the collector throws a `RuntimeException` if it collects a value greater than 2. The collection logic is wrapped in a `try`/`catch` statement because we don't want our program to crash and log the exception. The flow internally raises a `NumberformatException` if the value is negative. The `catch` operator acts as a safeguard (logs the exception and cancels the flow).

The output is:

```
Received 0
Collector stopped collecting the flow
```

Note that the flow didn't intercept the exception raised inside the collector, because it was caught in the catch clause of the `try`/`catch`. The flow never got to raise a `NumberformatException`, because the collector prematurely cancelled the collection.

Another example

In "Use Case #2: Concurrently Transform a Stream of Values", we held off on talking about error handling. Suppose the `transform` function might raise exceptions, among which is `NumberFormatException`. You can selectively handle `NumberFormatException` using the `catch` operator:

```
fun main() = runBlocking {
    // Defining the Flow of Content - nothing is executing yet
    val contentFlow = locationsFlow.map { loc ->
        flow {
```

```
                emit(transform(loc))
            }.catch { cause: Throwable ->
                if (cause is NumberFormatException) {    ❶
                    println("Handling $cause")
                } else {
                    throw cause                          ❷
                }
            }
        }.flattenMerge(4)

        // We now collect the entire flow using the toList terminal operator
        val contents = contentFlow.toList()
    }
```

❶ As the `catch` operator catches `Throwable`s, we need to check the type of the error. If the error is a `NumberFormatException`, then we handle it inside the `if` statement. You can add other checks there for different error types.

❷ Otherwise, you don't know the error's type. In most cases, it's preferable not to swallow the error and rethrow.

You can use emit from inside catch

Sometimes it will make sense to emit a particular value when you catch an exception from inside the flow:

You can find the corresponding source code in GitHub.

The output is:

```
Received 1
Received 3
Received 0
```

Emitting values from inside `catch` is especially useful to *materialize exceptions*.

Materialize Your Exceptions

Materializing exceptions[5] is the process of catching exceptions and emitting special values or objects that represent those exceptions. The goal is to avoid throwing exceptions from inside the flow, because code execution then goes to whatever place that collects that flow. It doesn't matter whether collection code handles exceptions thrown by the flow or not. If the flow throws exceptions, the collection code needs to be aware of those exceptions and catch them in order to avoid undefined behavior. Consequently, the flow has a *side effect on the collection code*, and this is a violation of the exception transparency principle.

NOTE

The collection code shouldn't be aware of implementation details of the flow. For example, if the flow is a `Flow<Number>`, you should only expect to get `Number` values (or subtypes)—not exceptions.

Let's take another example. Imagine you're fetching images, given their URLs. You have an incoming flow of URLs:

```
// We don't use realistic URLs, for brevity
val urlFlow = flowOf("url-1", "url-2", "url-retry")
```

You also have this function already available:

```
suspend fun fetchImage(url: String): Image {
    // Simulate some remote call
    delay(10)

    // Simulate an exception thrown by the server or API
    if (url.contains("retry")) {
        throw IOException("Server returned HTTP response code 503")
    }

    return Image(url)
}

data class Image(val url: String)
```

This `fetchImage` function may throw `IOException`s. In order to craft a "flow of images" using the `urlFlow` and the `fetchImage` function, you should materialize `IOException`s. Regarding the `fetchImage` function, it either succeeds or fails—you either get an `Image` instance, or an exception is thrown. You can represent these outcomes by a `Result` type, with `Success` and `Error` subclasses:[6]

```
sealed class Result
data class Success(val image: Image) : Result()
data class Error(val url: String) : Result()
```

In the case of a success, we wrap the actual result—the `Image` instance. In the case of failure, we felt it was appropriate to wrap the URL for which image retrieval failed. However, you're free to wrap all data that might be useful for the collection code, such as the exception itself.

Now you can encapsulate `fetchImage` usage, by creating a `fetchResult` function which returns `Result` instances:

```
suspend fun fetchResult(url: String): Result {
    println("Fetching $url..")
    return try {
        val image = fetchImage(url)
        Success(image)
    } catch (e: IOException) {
        Error(url)
    }
}
```

Finally, you can implement a `resultFlow` and collect it safely:

```
fun main() = runBlocking {
    val urlFlow = flowOf("url-1", "url-2", "url-retry")

    val resultFlow = urlFlow
        .map { url -> fetchResult(url) }
```

```
    val results = resultFlow.toList()
    println("Results: $results")
}
```

The output is:

```
Fetching url-1..
Fetching url-2..
Fetching url-retry..
Results: [Success(image=Image(url=url-1)), Success(image
```

A bonus

Imagine that you'd like to automatically retry fetching an image in the event of an error. You can implement a custom flow operator that retries an `action` while the `predicate` returns true:

```
fun <T, R : Any> Flow<T>.mapWithRetry(
    action: suspend (T) -> R,
    predicate: suspend (R, attempt: Int) -> Boolean
) = map { data ->
    var attempt = 0L
    var shallRetry: Boolean
    var lastValue: R? = null
    do {
        val tr = action(data)
        shallRetry = predicate(tr, ++attempt)
        if (!shallRetry) lastValue = tr
    } while (shallRetry)
    return@map lastValue
}
```

If you'd like to retry, three times (at most) before returning an error, you can use this operator like so:

```
fun main() = runBlocking {
    val urlFlow = flowOf("url-1", "url-2", "url-retry")

    val resultFlowWithRetry = urlFlow
        .mapWithRetry(
            { url -> fetchResult(url) },
            { value, attempt -> value is Error && attempt < 3L }
        )

    val results = resultFlowWithRetry.toList()
    println("Results: $results")
}
```

The output is:

```
Fetching url-1..
Fetching url-2..
Fetching url-retry..
Fetching url-retry..
Fetching url-retry..
Results: [Success(image=Image(url=url-1)), Success(image
```

Hot Flows with SharedFlow

Previous implementations of flow were *cold*: nothing runs until you start collecting the flow. This is made possible because for each emitted value, only one collector would get the value. Therefore, there's no need to run anything until the collector is ready to collect the values.

However, what if you need to *share* emitted values among several collectors? For example, say an event like a file download completes in your app. You might want to directly notify various components, such as some view-models, repositories, or even some views. Your file downloader might not have to be aware of the existence of other parts of your app. A good separation of concerns starts with a loose coupling of classes, and the *event bus* is one architecture pattern that helps in this situation.

The principle is simple: the downloader emits an event (an instance of a class, optionally holding some state) by giving it to the event bus, and all subscribers subsequently receive that event. A `SharedFlow` can act just like that, as shown in Figure 10-7.

Figure 10-7. `SharedFlow`.

A `SharedFlow` broadcasts events to all its subscribers. Actually, `SharedFlow` really is a toolbox that can be used in many situations—not just to implement an event bus. Before giving examples of usage, we'll show how to create a `SharedFlow` and how you can tune it.

Create a SharedFlow

In its simplest usage, you invoke `MutableSharedFlow()` with no parameter. As its name suggests, you can *mutate* its state, by sending values to it. A common pattern when creating a `SharedFlow` is to create a private mutable version and a public nonmutable one using `asSharedFlow()`, as shown in the following:

```
private val _sharedFlow = MutableSharedFlow<Data>()
val sharedFlow: SharedFlow<Data> = _sharedFlow.asSharedFlow()
```

This pattern is useful when you ensure that subscribers will only be able to *read* the flow (e.g., not send values). You might be surprised to find that `MutableSharedFlow` is not a class. It's actually a function that accepts parameters, which we'll cover later in this chapter. For now, we're only showing the default no-arg version of `MutableSharedFlow`.

Register a Subscriber

A subscriber registers when it starts collecting the `SharedFlow`—preferably the public nonmutable version:

```
scope.launch {
   sharedFlow.collect { data ->
      println(data)
   }
}
```

A subscriber can only live in a scope, because the `collect` terminal operator is a suspending function. This is good for structured concurrency: if the scope is cancelled, so is the subscriber.

Send Values to the SharedFlow

A `MutableSharedFlow` exposes two methods to emit values—`emit` and `tryEmit`:

`emit`
: This suspends under some conditions (discussed shortly).

`tryEmit`
: This never suspends. It tries to emit the value immediately.

Why are there two methods to emit values? This is because, by default, when a `MutableSharedFlow` emits a value using `emit`, it suspends until *all* subscribers start processing the value. We will give an example of `emit` usage in the next section.

However, sometimes this isn't what you want to do. You'll find situations where you have to emit values from nonsuspending code (see "Using SharedFlow as an Event Bus"). So here comes `tryEmit`, which tries to emit a value immediately and returns `true` if it succeeded, and `false` otherwise. We'll provide more details on the nuances of `emit` and `tryEmit` in upcoming sections.

Using SharedFlow to Stream Data

Suppose you are developing a news app. One of the features of your app is that it fetches news from an API or a local database and displays this news (or newsfeed). Ideally, you should rely on a local database to avoid using the API when possible. In this example, we'll use the API as the only source of news, although you can easily extend on our example to add local persistence.

The architecture

In our architecture, a view-model relies on a repository to get the newsfeed. When the view-model receives news, it notifies the view. The repository is responsible for querying the remote API at regular intervals, and provides a means for view-models to get the newsfeed (see Figure 10-8).

Figure 10-8. App architecture.

The implementation

To keep it simple, the following `News` data class represents news:

```
data class News(val content: String)
```

The repository reaches the API through a `NewsDao`. In our example, the data access object (DAO) is manually constructor-injected. In a real appli-

cation, we recommend that you use a dependency injection (DI) framework such as Hilt or Dagger:

```
interface NewsDao {
    suspend fun fetchNewsFromApi(): List<News>
}
```

We now have enough material to implement the repository:

```
class NewsRepository(private val dao: NewsDao) {
    private val _newsFeed = MutableSharedFlow<News>()                    ❶
    val newsFeed = _newsFeed.asSharedFlow()                              ❷

    private val scope = CoroutineScope(Job() + Dispatchers.IO)

    init {
        scope.launch {                                                   ❸
            while (true) {
                val news = dao.fetchNewsFromApi()
                news.forEach { _newsFeed.emit(it) }                      ❹

                delay(3000)
            }
        }
    }

    fun stop() = scope.cancel()
}
```

❶ We create our private mutable shared flow. It will only be used inside the repository.

❷ We create the public nonmutable version of the shared flow.

❸ As soon as the repository instance is created, we start fetching news from the API.

❹ Every time we get a list of `News` instances, we emit those values using our `MutableSharedFlow`.

All that's left is to implement a view-model that will subscribe to the repository's shared flow:

```
class NewsViewsModel(private val repository: NewsRepository) : ViewModel() {
    private val newsList = mutableListOf<News>()

    private val _newsLiveData = MutableLiveData<List<News>>(newsList)
    val newsLiveData: LiveData<List<News>> = _newsLiveData

    init {
        viewModelScope.launch {
            repository.newsFeed.collect {
                println("NewsViewsModel receives $it")
                newsList.add(it)
                _newsLiveData.value = newsList
            }
        }
    }
}
```

By invoking `repository.newsFeed.collect { .. }`, the view-model subscribes to the shared flow. Every time the repository emits a `News` instance to the shared flow, the view-model receives the news and adds it to its `LiveData` to update the view.

Notice how the flow collection happens inside a coroutine started with `viewModelScope.launch`. This implies that if the view-model reaches its end-of-life, the flow collection will automatically be cancelled, and that's a good thing.

TIP

In our example, we manually constructor-inject an object (in this case, the repository). A DI framework would definitely help to avoid boilerplate code. As demonstrating DI frameworks isn't the primary focus of this chapter, we chose to go for a manual repository injection into the view-model.

Test of our implementation

In order to test the previous code, we need to mock the `NewsDao`. Our DAO will just send two dummy `News` instances and increment a counter:

```
val dao = object : NewsDao {
    private var index = 0

    override suspend fun fetchNewsFromApi(): List<News> {
        delay(100)  // simulate network delay
        return listOf(
            News("news content ${++index}"),
            News("news content ${++index}")
        )
    }
}
```

When we run our code using the preceding DAO, this is what we see in the console:

```
NewsViewsModel receives News(content=news content 1)
NewsViewsModel receives News(content=news content 2)
NewsViewsModel receives News(content=news content 3)
...
```

There is nothing surprising here: our view-model simply receives the news sent by the repository. Things become interesting when there's not one but several view-models that subscribe to the shared flow. We've gone ahead and created another view-model which also logs in the console. We created the other view-model 250 ms *after* the launch of the program. This is the output we get:

```
NewsViewsModel receives News(content=news content 1)
NewsViewsModel receives News(content=news content 2)
NewsViewsModel receives News(content=news content 3)
AnotherViewModel receives News(content=news content 3)
NewsViewsModel receives News(content=news content 4)
AnotherViewModel receives News(content=news content 4)
NewsViewsModel receives News(content=news content 5)
AnotherViewModel receives News(content=news content 5)
```

```
NewsViewsModel receives News(content=news content 6)
AnotherViewModel receives News(content=news content 6)
...
```

You can see that the other view-model *missed* the first two news entries. This is because, at the time the shared flow emits the first two news entries, the first view-model is the only subscriber. The second view-model comes after and only receives subsequent news.

Replay values

What if you need the second view-model to get previous news? A shared flow can *optionally* cache values so that new subscribers receive the last *n* cached values. In our case, if we want the shared flow to replay the last two news entries, all we have to do is to update the line in the repository:

```
private val _newsFeed = MutableSharedFlow<News>(replay = 2)
```

With that change, the two view-models receive *all* news. Replaying data is actually useful in other common situations. Imagine the user leaves the fragment that displays the list of news. Potentially, the associated view-model might also get destroyed, if its lifecycle is bound to the fragment (that wouldn't be the case if you chose to bound the view-model to the activity). Later on, the user comes back to the news fragment. What happens then? The view-model is re-created and immediately gets the last two news entries while waiting for fresh news. Replaying only two news entries might then be insufficient. Therefore, you might want to increase the replay count to, say, 15.

Let's recap. A `SharedFlow` can optionally replay values for new subscribers. The number of values to replay is configurable, using the `replay` parameter of the `MutableSharedFlow` function.

Suspend or not?

There's one last thing about this replay feature that you should be aware of. A shared flow with `replay` > `0` internally uses a cache that works similarly to a `Channel`. For example, if you create a shared flow with `replay` = `3`, the first three `emit` calls won't suspend. In this case, `emit` and `tryEmit` do exactly the same thing: they add a new value to the cache, as shown in Figure 10-9.

Figure 10-9. Replay cache not full.

When you submit a fourth value to the shared flow, then it depends on whether you use `emit` or `tryEmit`, as shown in Figure 10-10. By default, when the replay cache is full, `emit` suspends until all subscribers start processing the oldest value in the cache. As for `tryEmit`, it returns `false` since it can't add the value to the cache. If you don't keep track of that fourth value yourself, this value is lost.

Figure 10-10. Replay cache full.

That behavior (when the replay cache is full) can be changed. You can also opt to discard either the oldest value in the cache or the value that is being added to the cache. In both cases, `emit` does not suspend and `tryEmit` returns true. Therefore, there are three possible behaviors on buffer overflow: suspend, drop oldest, and drop latest.

You apply the desired behavior while creating the shared flow, by using the `onBufferOverflow` parameter, as shown in the following:

```
MutableSharedFlow(replay = 3, onBufferOverflow = BufferOverflow.DROP_OLDEST)
```

`BufferOverflow` is an *enum* with three possible values: `SUSPEND`, `DROP_OLDEST`, and `DROP_LATEST`. If you don't specify a value for `onBufferOverflow`, `SUSPEND` is the default strategy.

Buffer values

In addition to being able to replay values, a shared flow can *buffer* values without replaying them, allowing slow subscribers to lag behind other, faster subscribers. The size of the buffer is customizable, as shown in the following:

```
MutableSharedFlow(extraBufferCapacity = 2)
```

By default, `extraBufferCapacity` equals zero. When you set a strictly positive value, `emit` doesn't suspend while there is buffer space remaining—unless you explicitly change the buffer overflow strategy.

You might be wondering in what situations `extraBufferCapacity` can be useful. One immediate consequence of creating a shared flow with, for example, `extraBufferCapacity = 1` and `onBufferOverflow = BufferOverflow.DROP_OLDEST`, is that you're guaranteed that `tryEmit` will *always* successfully insert a value into the shared flow. It's sometimes really convenient to insert values in a shared flow from non-suspending code. A good example of such a use case is when using a shared flow as an event bus.

Using SharedFlow as an Event Bus

You need an event bus when all the following conditions are met:

- You need to broadcast an event across one or several subscribers.
- The event should be processed *only once*.
- If a component isn't registered as a subscriber at the time you emit the event, the event is lost for that component.

Notice the difference with `LiveData`, which keeps in memory the last emitted value and replays it every time the fragment is re-created. With an event bus, the fragment would only receive the event *once*. For example, if the fragment is re-created (the user rotates the device), the event won't be processed again.

An event bus is particularly useful when you want, for example, to display a message as a `Toast` or `Snackbar`. It makes sense to display the message only once. To achieve this, a repository can expose a shared flow

as shown in the following code. In order to make the exposed flow accessible for view-models, or even fragments, you can use a DI framework such as Hilt or Dagger:

```
class MessageRepository {
    private val _messageFlow = MutableSharedFlow<String>(
        extraBufferCapacity = 1,
        onBufferOverflow = BufferOverflow.DROP_OLDEST
    )
    val messageEventBus = _messageFlow.asSharedFlow()

    private fun someTask() {
        // Notify subscribers to display a message
        _messageFlow.tryEmit("This is important")
    }
}
```

We've set `extraBufferCapacity` to 1 and `onBufferOverflow` to `DROP_OLDEST` so that `_messageFlow.tryEmit` always emits successfully. Why do we care about `tryEmit`? In our example, we use `_messageFlow` from a nonsuspending function. Therefore, we can't use `emit` inside `someTask`.

If you use `_messageFlow` from inside a coroutine, you can use `emit`. The behavior would be exactly the same, since `emit` wouldn't suspend because of the presence of the buffer and the buffer overflow policy.

An event bus is appropriate for dispatching one-time events that some components might miss if they're not ready to receive those events. For example, say you fire a "recording-stopped" event while the user hasn't navigated to the fragment displaying recordings yet. The result is that the event is lost. However, your application can be designed to update the state of the fragment anytime the fragment resumes. Consequently, receiving "recording-stopped" is only useful when the fragment is in the resumed state, as this should trigger a state update. This is just an example of when losing events is totally acceptable and part of your application's design.

Sometimes, however, this isn't what you want to achieve. Take, for example, a service that can perform downloads. If the service fires a "download-finished" event, you don't want your UI to miss that. When the user navigates to the view displaying the status of the download, the view should render the updated *state* of the download.

You will face situations where sharing a *state* is required. This situation is so common that a type of shared flow was specifically created for it: `StateFlow`.

StateFlow: A Specialized SharedFlow

When sharing a state, a state flow:

- Shares only one value: the current *state*.
- Replays the state. Indeed, subscribers should get the last state even if they subscribe afterward.
- Emits an initial value—much like `LiveData` has an initial value.
- Emits new values only when the state changes.

As you've learned previously, this behavior can be achieved using a shared flow:

```
val shared = MutableSharedFlow(
    replay = 1,
    onBufferOverflow = BufferOverflow.DROP_OLDEST
)
shared.tryEmit(initialValue) // emit the initial value
val state = shared.distinctUntilChanged() // get StateFlow-like behavior
```

`StateFlow`[7] is a shorthand for the preceding code. In practice, all you have to write is:

```
val state = MutableStateFlow(initialValue)
```

An Example of StateFlow Usage

Imagine that you have a download service that can emit three possible download states: download started, downloading, and download finished, as shown in Figure 10-11.

Figure 10-11. Download state.

Exposing a flow from an Android service can be done in several ways. If you need high decoupling for, say, testability purposes, a DI-injected "repository" object can expose the flow. The repository is then injected in all components that need to subscribe. Or the service can statically expose the flow in a companion object. This induces tight coupling between all components that use the flow. However, it might be acceptable in a small app or for demo purpose, such as in the following example:

```
class DownloadService : Service() {
    companion object {
        private val _downloadState =
            MutableStateFlow<ServiceStatus>(Stopped)
        val downloadState = _downloadState.asStateFlow()
    }
    // Rest of the code hidden for brevity
}

sealed class ServiceStatus
object Started : ServiceStatus()
data class Downloading(val progress: Int) : ServiceStatus()
object Stopped : ServiceStatus()
```

Internally, the service can update its state by using, for example, `_downloadState.tryEmit(Stopped)`. When declared inside a companion object, the state flow can be easily accessed from a view-model, and exposed as a `LiveData` using `asLiveData()`:

```
class DownloadViewModel : ViewModel() {
    val downloadServiceStatus = DownloadService.downloadState.asLiveData()
}
```

Subsequently, a view can subscribe to the `LiveData`:

```
class DownloadFragment : Fragment() {
    private val viewModel: DownloadViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        viewModel.downloadServiceStatus.observe(this) {  (1)
            it?.also {
                onDownloadServiceStatus(it)
            }
        }
    }

    private fun onDownloadServiceStatus(
        status: ServiceStatus
    ): Nothing = when (status) {  (2)
        Started -> TODO("Show download is about to start")
        Stopped -> TODO("Show download stopped")
        is Downloading -> TODO("Show progress")
    }
}
```

(1) We subscribe to the `LiveData`. If we receive a nonnull value, then we invoke `onDownloadServiceStatus` method.

(2) We are purposely using `when` as an expression so that the Kotlin compiler guarantees that all possible types of `ServiceStatus` are taken into account.

You might be wondering why we used a state flow, and why we haven't used a `LiveData` in the first place—eliminating the need of `asLiveData()` in the view-model.

The reason is simple. `LiveData` is Android-specific. It's a lifecycle-aware component which is meaningful when used within Android views. You might design your application with Kotlin multiplatform code in mind. When targeting Android and iOS, only multiplatform code can be shared as common code. The coroutine library is multiplatform. `LiveData` isn't.

However, even when not considering Kotlin multiplatform, the Flows API makes more sense since it provides greater flexibility with all its flows operators.

Summary

- The Flows API allows for *asynchronous data stream transformation*. A lot of operators are already available out of the box and cover most use cases.
- Thanks to the *composable* nature of flow operators, you can fairly easily design your own, if you need to.
- Some parts of the flow can be offloaded to a background thread or thread pool, and yet keep a high-level view of data transformation.
- A shared flow broadcasts values to all its subscribers. You can enable buffering and/or replay of values. Shared flows really are a toolbox. You can use them as an event bus for one-time events, or in more complex interactions between components.

- When a component shares its state, a special kind of shared flow is appropriate for use: state flow. It replays the last state for new subscribers and only notifies subscribers when the state changes.

1 We'll refer to `Flow`s as *flows* in the rest of this chapter.

2 A token is generally encrypted registration data which the client application stores in memory so that further database access doesn't require explicit authentication.

3 As opposed to cold, a hot entity lives on its own until explicitly stopped.

4 The coroutine started with `coroutineScope{}`.

5 *Materialize* comes from the Rx operator of the same name. See the Rx documentation for more insight.

6 These subclasses are an algebraic data type.

7 Actually, `StateFlow` *is* a `SharedFlow` under the hood.