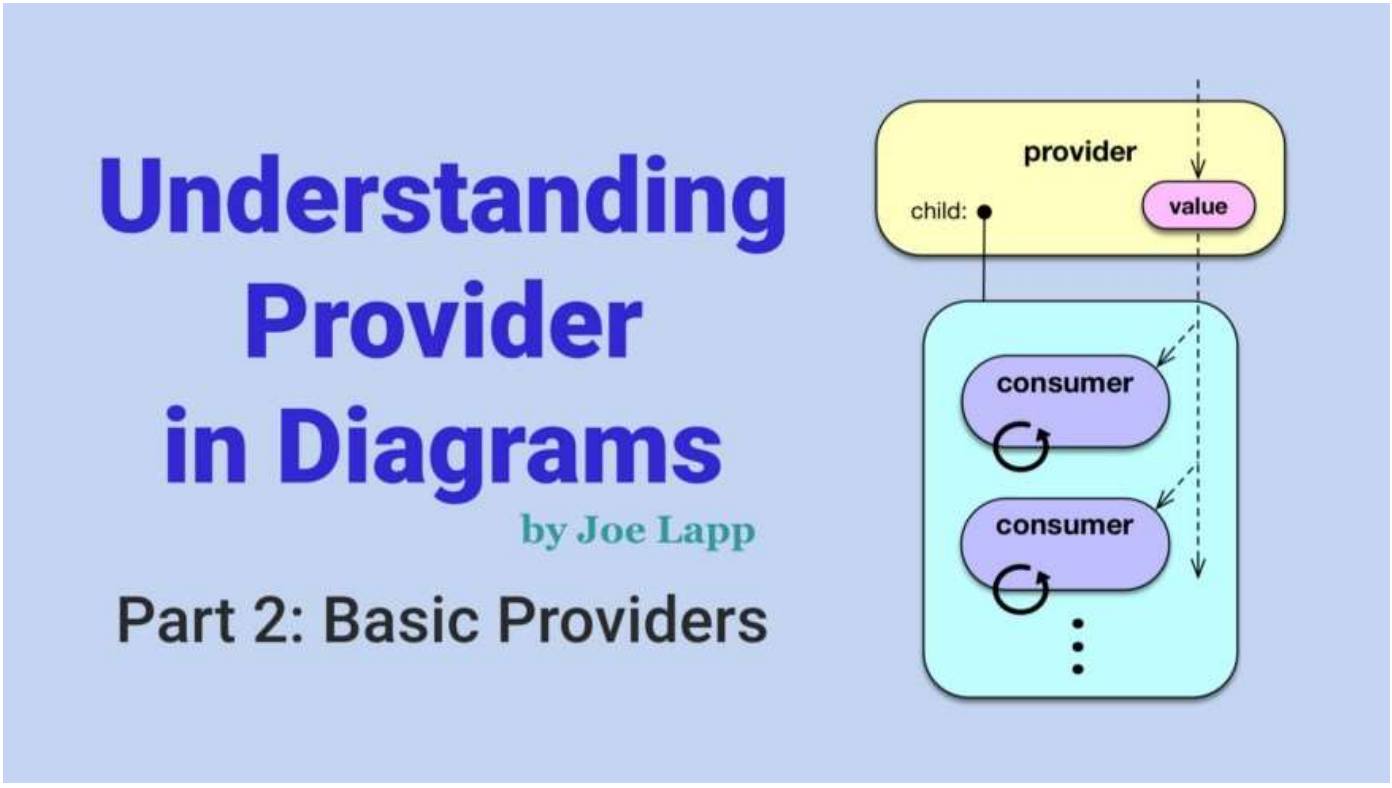
Published in Flutter Community

Joseph T. Lapp Follow

Oct 4, 2019 · 12 min read · Listen

Save

Understanding Provider in Diagrams — Part 2: Basic Providers



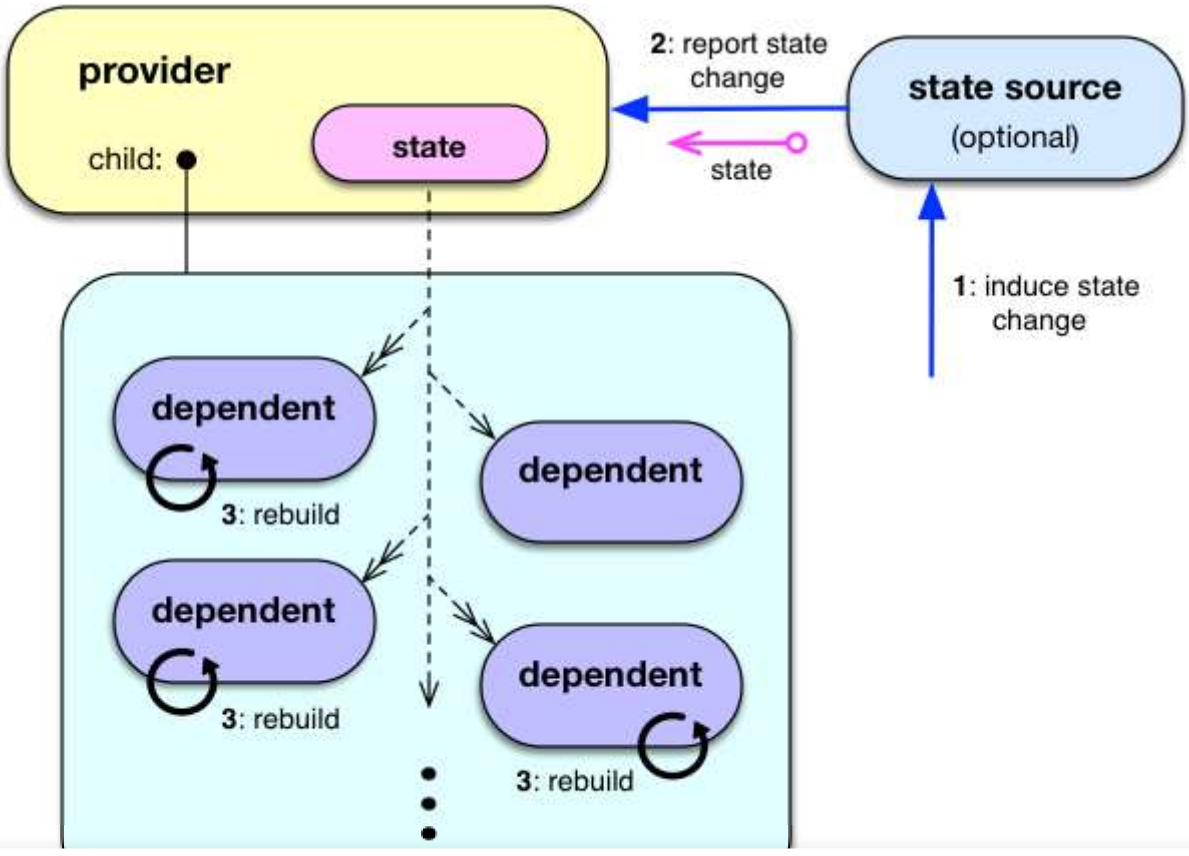
This article is the second in a three-part series that describes the architecture of the Flutter provider package and illustrates this architecture in diagrams. It assumes the reader has read the first article, Part 1: Providing values. This second article introduces the basic kinds of providers. See also Part 3: Architecture.

design your own specialized provider. You can choose the provider having the efficiency you need and the economy of expression you desire. Now that you are familiar with how providers provide values and rebuild dependents, let's get a good sense of the basic kinds of providers.

State Sources and their Providers

"State" is nothing but a value. If we're going to rebuild widgets on state changes, we need something that reports changes in state values. The provider package does not have a term for objects that do this, so we'll call them "state sources." A state source, then, is an object that reports state changes. State sources include streams, futures, and instances of Listenable, among others. The provider package includes a variety of providers specialized for subscribing to various kinds of state sources.

The following diagram generically illustrates the process of rebuilding dependent widgets when a state source reports a state change:



state value by the time of this first build, the provider hands the dependents an application-supplied default initial state value. Only three dependents in this diagram subsequently rebuild on state changes.

The process of rebuilding dependents begins with a state change, as follows:

1. Something induces a new state in the state source, such as an external event precipitating the change or an external object sending a new state value. For example, the above dependent that doesn't rebuild might represent a button whose pressing causes a state change.
2. The state source informs the provider that the state has changed. This may also entail the provider retrieving the new state value from the state source, depending on the kind of state source. The provider, via its internal InheritedWidget, marks for building the dependent widgets that are listening for changes in the provider's value.
3. The Flutter framework rebuilds the dependents previously marked for building, in the process acquiring the new state value as explained previously.

Let's make this more concrete by examining the basic kinds of providers.

Basic Kinds of Providers

Different providers interact differently with state sources to inform their dependents of state changes. After examining the basic kinds of providers, we'll finally be in position to understand the entire architecture diagram.

StreamProvider

A StreamProvider receives events from a stream and provides these events to its dependent widgets as values representing state. Use a provider of type `StreamProvider<T>` to receive values of type `T` from a `Stream<T>`.

A StreamProvider is said to "subscribe" to a stream to receive the events that the

dependents with the most recently emitted event. See the StreamProvider API for options governing streams.

Here's an example of a StreamProvider that emulates loading data:

```
StreamProvider<int>(
  initialData: 0,
  builder: (context) {
    // Pretend this is loading data and reporting percent loaded.
    return Stream<int>
      .periodic(Duration(milliseconds: 100), (count) => count + 1)
      .take(100);
  },
  child: ...
}
```

And here's an example of a dependent that uses this StreamProvider:

```
Consumer<int>(
  builder: (context, percentDone, child) {
    if (percentDone < 100) {
      return Text("Loading... ($percentDone% done)");
    }
    return Text("Done loading!");
  },
),
```

You'll find the complete source for this example Stream app here.

To establish a StreamProvider from a StreamController, use the `StreamProvider<T>.controller` constructor instead. It's necessary to use this latter constructor if you want the provider to close the stream when the provider is disposed; `StreamProvider<T>` never closes the stream.

FutureProvider

This value has type `T` and represents state.

A FutureProvider is said to "subscribe" to a future to receive the value of the future at completion. Prior to completion, the provider provides its dependents with an initial value (also of type `T`), supplied via the `initialData` constructor parameter. Upon completion, the provider provides its dependents with the value of the completed future. There are therefore only two values available to the dependents — the initial value and the value at completion — and consequently, the dependents rebuild at most once.

Here's an example of a FutureProvider that emulates saving data:

```
FutureProvider<bool>(
  initialData: true,
  builder: (context) {
    // Pretend we're saving data and it takes 4 seconds.
    return Future.delayed(Duration(seconds: 4), () => false);
  },
  child: ...
),
```

And here's an example of a dependent that uses this FutureProvider:

```
Consumer<bool>(
  builder: (context, saving, child) {
    return Text(saving ? "Saving..." : "Saved!");
  },
),
```

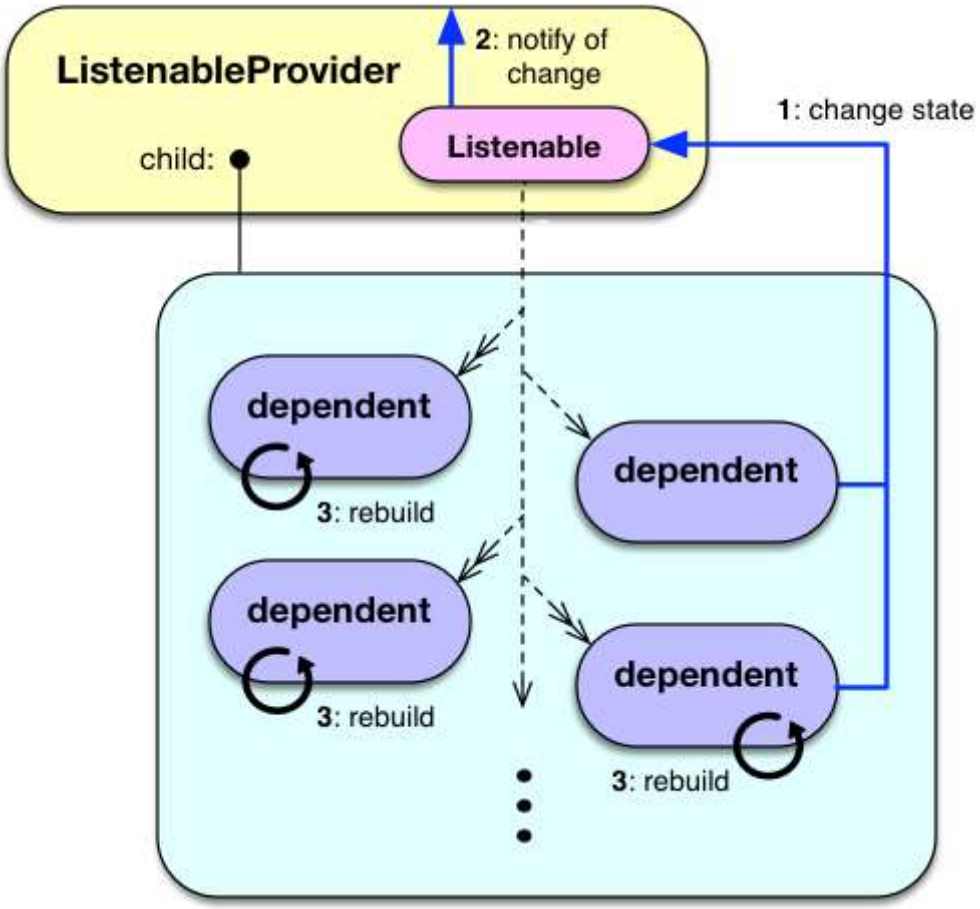
You'll find the complete source for this example Future app here.

ListenableProvider and ChangeNotifierProvider

ListenableProvider and its subclass ChangeNotifierProvider are different from the

provider of a state change, the provider provides its dependents with the Listenable itself as the value. Dependents treat the Listenable as a model, accessing its members to get the state they need.

The following diagram illustrates the process of rebuilding for state change when using a ListenableProvider or ChangeNotifierProvider:



The process of rebuilding dependents is as follows:

1. Something causes a change of state in the Listenable. The diagram depicts one of two dependent widgets causing this change. For example, a dependent widget might be a button that changes the state when pressed. The button widget needs to access the Listenable to change its state. It gets the Listenable either by calling `Provider.of<T>()` or by receiving it as a parameter of the `builder` function

listening dependent widgets for building.

3. The Flutter framework rebuilds the dependents that are marked for building, acquiring the provided value in the process, as explained previously. However, in this case, the Listenable is the provided value — the same Listenable gets handed to the dependents on every rebuild. The dependents access the Listenable for the state they need for building.

The state source subclasses (or mixes in) Listenable or ChangeNotifier. Use `ListenableProvider<T>` for a `T` that inherits from Listenable, and use `ChangeNotifierProvider<T>` for a `T` that inherits from ChangeNotifier. Technically, you could use a ChangeNotifier with `ListenableProvider<T>`, but `ChangeNotifierProvider<T>` provides the additional benefit of disposing the ChangeNotifier when done using ChangeNotifier's `dispose()` method.

Here's an example of a ChangeNotifier that implements a counter:

```
class Counter with ChangeNotifier {
  int count = 0;

void increment() {
    ++count;
    notifyListeners();
  }
}
```

You can make this state source available to dependents as follows:

```
ChangeNotifierProvider<Counter>(
  builder: (context) => Counter(),
  child: ...
),
```


```
      '${counter.count}',
      style: Theme.of(context).textTheme.display1,
    ),
  ),
```

And here's a dependent widget that only builds once because it needs a reference to the state source but doesn't change with changing state:

```
Builder(builder: (context) {
  final counter = Provider.of<Counter>(context, listen: false);
  return RaisedButton(
    onPressed: () => counter.increment(),
    child: Text("Increment"),
  );
}),
```

As you can see, when the state source is the provided value, the state source actually serves as a model. The model changes state internally and notifies the provider when it has changed. The provider relays this notification to its dependents and provides the dependents with access to the model.

You'll find the complete source for this example ChangeNotifier app here.

ValueListenableProvider

Despite the name, a ValueListenableProvider is not a kind of ListenableProvider. It works more like a StreamProvider or a FutureProvider, even though its state source, a ValueListenable, is a kind of Listenable. ValueListenable is actually an interface for which Flutter provides two implementations: Animation and ValueNotifier.

A ValueListenableProvider subscribes to a ValueListenable to receive value change notifications. Upon receiving a change notification, it retrieves the value from the ValueListenable and provides this value to its dependents as state. Use `ValueListenableProvider<T>` to receive values of type `T` from a `ValueListenable<T>`.

no `initialData` parameter is necessary.

Here's an example of a ValueListenable that implements a countdown:

```
class CountDown extends ValueNotifier<int> {
  CountDown(int downFrom) : super(downFrom) {
    scheduleDecrement();
  }

void scheduleDecrement() {
    Future.delayed(Duration(seconds: 1), () {
      if (--value > 0) scheduleDecrement();
    });
  }
}
```

You can make this state source available to dependents as follows:

```
ValueListenableProvider<int>(
  builder: (context) => CountDown(10),
  child: ...
),
```

The dependent listens to the value as follows:

```
Consumer<int>(
  builder: (context, count, child) {
    if (count > 0) {
      return Text("T minus $count seconds");
    }
    return Text("Blast off!");
  },
),
```

with StreamProvider.[1])

You'll find the complete source for this example ValueNotifier app here.

Note that when using the constructor `ValueListenableProvider<T>`, the state source *must* be a `ValueNotifier` and not just any `ValueListenable`. That's because this version of the provider calls the `dispose()` method available on the `ValueNotifier` when the provider is destroyed. The state source can be any `ValueListenable` when using the `ValueListenableProvider<T>.value` constructor. We'll look at `.value` providers shortly.

Plain Vanilla Provider

The Provider class implements the most basic kind of provider. We might call the "plain vanilla provider." This provider simply makes a value available to descendent widgets, without subscribing the provider to the value. There are no restrictions on the kind of value. However, when the value is a state source, mechanisms outside the provider are responsible for handling state change.

The provider README includes an example demonstrating use of a plain vanilla provider with both a state value and a state source. A StatefulWidget employs a State that contains a `_count` variable. We duplicate the State code here:

```
class ExampleState extends State<Example> {
  int _count;

  void increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Provider.value(
      value: _count,
      child: Provider.value(
```

Notice that the `build()` method returns a Provider that contains a nested Provider. The outer Provider exposes a value of type integer to dependent widgets. The nested Provider exposes the State object itself to dependent widgets. (For the moment, ignore the fact that it's using `.value` constructors — we look at these in the next section.)

In this example, the State object serves as the state source for an integer state value. It has an `increment()` method for changing the state. Changing the state results in rebuilding both providers. However, because the inner `child` references a pre-existing widget, the inner `child` does not automatically rebuild; the dependents only rebuild when their dependent state changes.

Dependent widgets retrieve the state value by calling `Provider.of<int>()`, and they retrieve the state source by calling `Provider.of<ExampleState>()`. When a dependent calls `increment()` on ExampleState, the state value changes, and all of the dependents listening to values of type `int` rebuild. However, because the value of type ExampleState never changes (remains the same instance), none of the dependents of ExampleState rebuild, regardless of whether they are listening for changes using `listen: true`.

.value Providers vs. Disposing Providers

There are at least two versions of every kind of provider. One version is responsible for creating and disposing the state source. This version "owns" the state source and manages its lifetime. That's the version we've introduced so far. Let's call them "disposing providers." The other version only references the state source and does not manage its lifetime. This version of a provider has a constructor ending in `.value`. We call them "`.value` providers".

Suppose we wish to use a ChangeNotifierProvider with a Counter class that implements ChangeNotifier. To have the provider manage the lifetime of the Counter,

```
  child: ...
),
```

This defers the construction of the state source to the provider. This is “lazy” construction in the sense that the provider doesn’t construct the state source until Flutter gets around to building the provider. Once constructed, the provider uses this instance of state source until the provider is destroyed. At this point, to release the state source’s resources, the provider calls the `dispose()` or `close()` method on the state source, as required.

Contrast this with `.value` providers, which do not manage the state source lifetime. In our example of a Counter that implements ChangeNotifier, suppose we create the instance of Counter outside of the provider and want to use this instance in the provider. We would use the `.value` constructor:

```
ChangeNotifierProvider<Counter>.value(
  value: counter,
  child: ...
),
```

In disposing providers, the `builder` parameter provides a function for creating the state source. In `.value` providers, there is instead a `value` parameter, which takes a reference to the state source. Take care not to confuse the `builder` parameter of a provider with the `builder` parameter of a Builder widget: the former creates a state source, the latter creates a Widget.

The disposing version of the plain vanilla provider allows you to hand the constructor a `dispose` function for disposing the state source. The provider calls this function when it itself is disposed.

Disposing providers are generally recommended so that you need not worry about

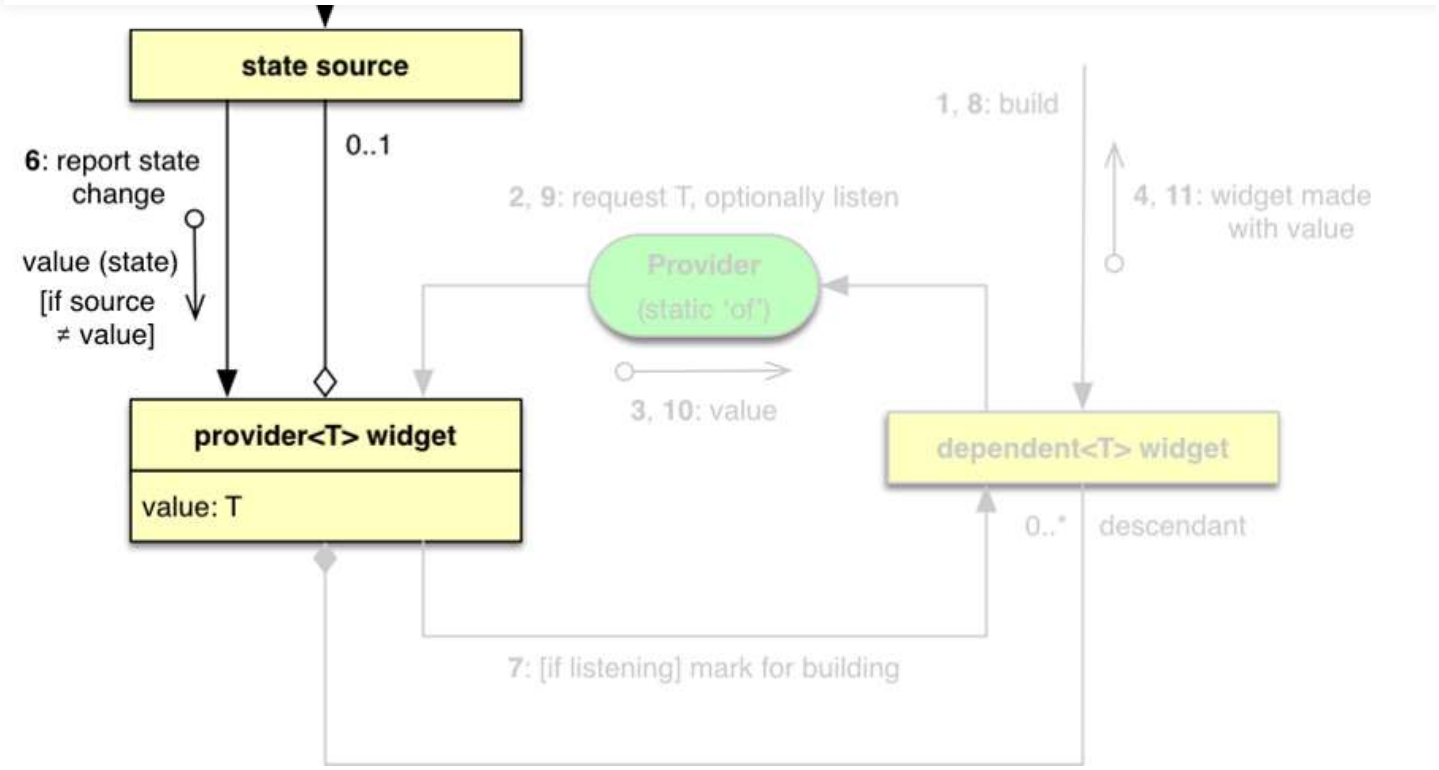
2. You have a state variable in the State of a StatefulWidget that you want to make available to descendent widgets.

3. You have a dependent widget that has retrieved an object value from an ancestor provider, and you want to make properties of this object available to descendent widgets via the property's type. For example, you might have a dependent that has received state of type Counter, and you want to make `counter.count` available to descendants that ask for a state of type integer, knowing nothing about Counter.

A good rule of thumb is to use the disposing provider (without `.value`) whenever possible. When using a `.value` provider, you are responsible for creating and disposing the state source as needed.

Architecture for Sourcing State

We have now covered the final portion of the architecture diagram. All that was left was the means by which providers acquire state from state sources. Let's review this process, highlighted here in the UML diagram:



As we saw with the plain vanilla provider, a provider need not have a state source. It could merely expose unchanging values. The diagram depicts this by indicating that the state source is optional.

When there is a state source, the provider is either a disposing provider that owns it or a `.value` provider that references it.

The provider subscribes to the state source to receive state change notices. Some state sources provide the new state value along with the notice. Futures and streams are examples of such state sources. Other state sources, such as ValueListenable (including ValueNotifier), only provide the notice. In this latter case, the provider retrieves the state value from the state source.

When using a ChangeNotifier, or when using a Listenable with a ListenableProvider, the state source itself is the value provided to dependents. In this case, the dependents themselves read state directly from the state source, or they directly access the state source to change its state.

package, and explain why one would use the provider package instead of just using `InheritedWidget`.

Congratulations! *You've made it through Part 2 of Understanding Provider in Diagrams. This was the toughest of the three parts. Once you've recovered, please head on over to Part 3: Architecture, where we mainly summarize everything.*

[1] It's not clear to me when one would choose to use a ValueListenableProvider over a StreamProvider, unless the ValueListenable is an Animation. Perhaps the main benefit is that a ValueListenable can provide the initial value, better separating the business logic from the UI.

https://www.twitter.com/FlutterCom