Making sense of all those Flutter Providers

medium.com/flutter-community/making-sense-all-of-those-flutter-providers-e842e18f45dd

June 21, 2020

Although the official Flutter site (in *Simple app state management*) says that the Provider package "is easy to understand," I haven't found that to always be the case. I think the main cause is the sheer number of Provider types there are:

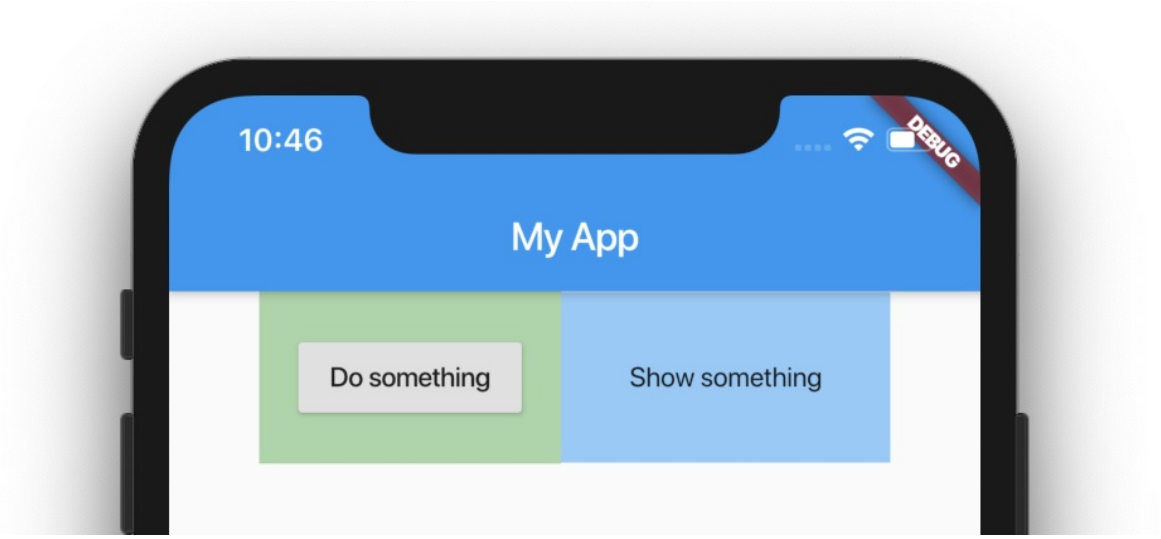
- Provider
- ListenableProvider
- ChangeNotifierProvider
- ValueListenableProvider
- StreamProvider
- FutureProvider
- MultiProvider
- ProxyProvider
- ChangeNotifierProxyProvider
- and more

I just want to manage my app state in an easy way. Why are there so many choices? Which one am I supposed to use? Where do I start?

The purpose of this article is to help you understand what each of the main Provider types are for. I'll give a minimal example of how each one is used. Then when you understand the differences, you can decide for yourself if and how you want to use the Provider package to manage app state in your project.

Setup

I'm going to use the following layout in the examples below:



This is the meaning:

- The “Do something” button represents any app event that changes the app state.
- The “Show something” Text widget represents any part of the UI that needs to display the app state.
- The green rectangle on the left and the blue rectangle on the right represent two different parts of the widget tree. They are there to emphasize that an event and the UI it updates may be in any part of the app.

Here is the code: code examples can be found in - developer_quest/notes/flutter_points/flutter_articles/blogs/provider_package_blogs

The examples below will also assume that you already have the provider package in your *pubspec.yaml* file:

```
dependencies:
  provider: ^4.0.1
```

And that you are importing it where needed:

```
import 'package:provider/provider.dart';
```

Provider

As you might imagine, `Provider` is the most basic of the Provider widget types. You can use it to provide a value (usually a data model object) to anywhere in the widget tree. However, it won’t help you update the widget tree when that value changes.

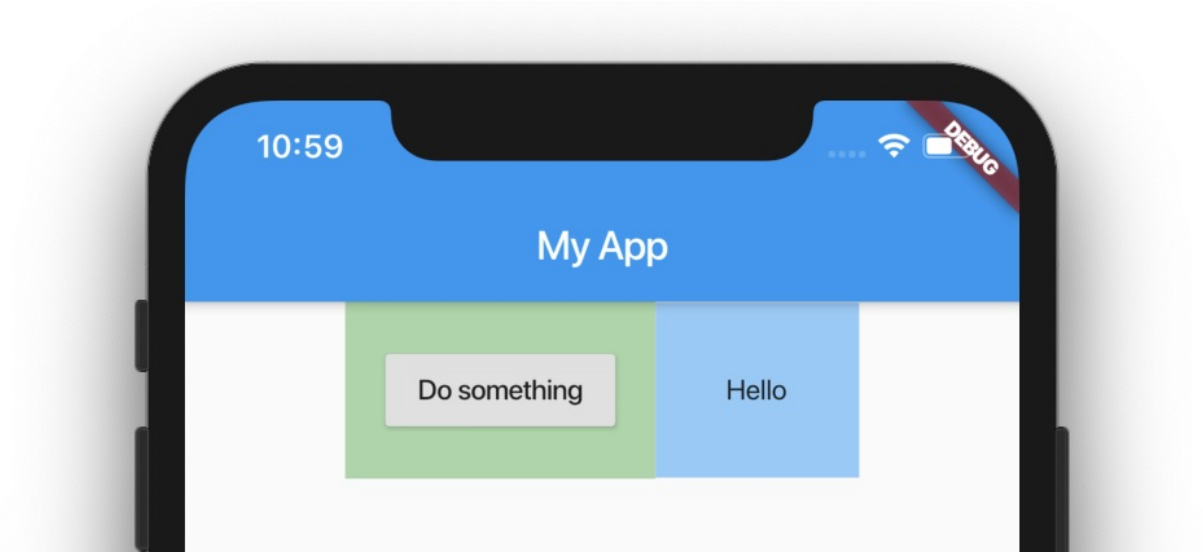
Say your app state is in a model like this:

```
class MyModel {
  String someValue = 'Hello';  void doSomething() {
    someValue = 'Goodbye';
    print(someValue);
  }
}
```

You can provide that model to the widget tree by wrapping the top of the tree with the `Provider` widget. You can get a reference to the model object by using the `Consumer` widget.

Find the `Provider` and two `Consumer` widgets in the code below:

Running this gives the following result:



Notes:

- The UI was built with the “Hello” text that came from the model.
- Pressing the “Do something” button will cause an event to happen on the model. However, even though the model’s data got changed, the UI wasn’t rebuilt because the `Provider` widget doesn’t listen for changes in the values it provides.

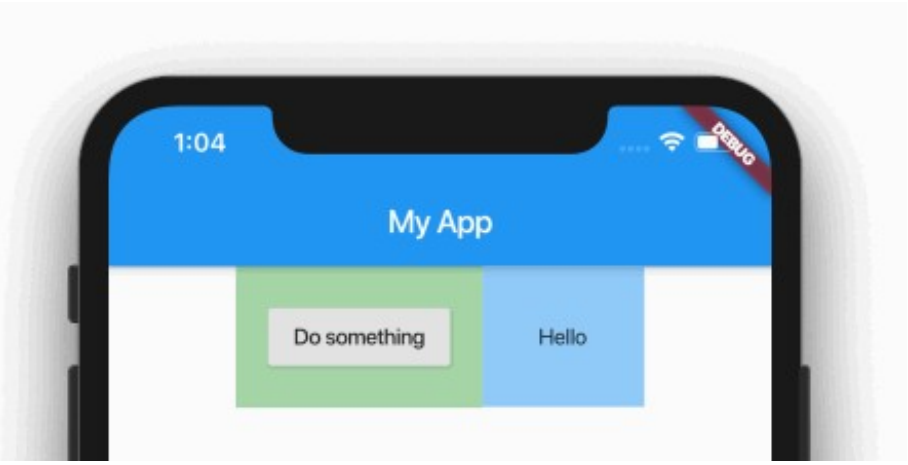
ChangeNotifierProvider

Unlike the basic `Provider` widget, `ChangeNotifierProvider` listens for changes in the model object. When there are changes, it will rebuild any widgets under the `Consumer`.

In the code, change `Provider` to `ChangeNotifierProvider`. The model class needs to use the `ChangeNotifier` mixin (or extend it). This gives you access to `notifyListeners()` and any time you call that, the `ChangeNotifierProvider` will be notified and the Consumers will rebuild their widgets.

Here is the complete code:

Now when you press the “Do something” button, the text changes from “Hello” to “Goodbye”.



Notes:

- In most apps your model class will be in its own file and you'll need to import `flutter/foundation.dart` in order to use `ChangeNotifier`. I'm not really a fan of that because that means your business logic now has a dependency on the framework, and the framework is a detail. But I'm willing to live with it for now.
- The `Consumer` widget rebuilds any widgets below it whenever `notifyListeners()` gets called. The button doesn't need to get updated, though, so rather than using a `Consumer`, you can use `Provider.of` and set the listener to false. That way the button won't be rebuilt when there are changes. Here is the button extracted into its own widget:

```
class MyButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {    final myModel = Provider.of<MyModel>(context,
listen: false);    return RaisedButton(
      child: Text('Do something'),
      onPressed: () {
        myModel.doSomething();
      },
    );
  }
}
```

In the examples below I'm just going to leave the button as it was, though. That is, using the `Consumer`.

FutureProvider

`FutureProvider` is basically just a wrapper around the `FutureBuilder` widget. You give it some initial data to show in the UI and also provide it a Future of the value that you want to provide. The `FutureProvider` listens for when the Future completes and then notifies the Consumers to rebuild their widgets.

In the code below I used an empty model to give some initial data to the UI. I also added a function to return a new model after 3 seconds. This is what the `FutureProvider` waits for.

Like the basic `Provider`, `FutureProvider` does not listen for any changes within the model itself. I show that in the code below by making the "Do something" button change the model after 2 seconds. There is no effect on the UI.

Here is the full code:

Notes:

- The `FutureProvider` tells the `Consumer` to rebuild after `Future<MyModel>` completes.
- Press hot restart to rebuild the app with initial values.
- Note that pressing the "Do something" button does not update the UI, even after the Future completes. If you want that kind of functionality, then just use the `ChangeNotifierProvider` from the last section.
- Your use case for `FutureProvider` might be to read some data from a file or the network. But you could also do that with a `FutureBuilder`. In my unexpert opinion, `FutureProvider` is not significantly more useful than a `FutureBuilder`. If I need a provider then I would probably use a `ChangeNotifierProvider`, and if I don't need a provider then I would probably use a `FutureBuilder`. I'm happy to update this if you would like to add a comment, though.

StreamProvider

`StreamProvider` is basically a wrapper around a `StreamBuilder`. You provide a stream and then then the Consumers get rebuilt when there is an event in the stream. The setup is very similar to the `FutureProvider` above.

You should consider the values that are emitted from the stream to be immutable. That is, the `StreamProvider` doesn't listen for changes in the model itself. It only listens for new events in the stream.

The code below shows a `StreamProvider` that provides a stream of model objects. Here is the full code:

Notes:

- The `StreamProvider` tells the `Consumer` to rebuild after when there is a new stream event.
- Press hot restart to rebuild the app with initial values.
- Note that pressing the "Do something" button does not update the UI. If you want that kind of functionality, then just use a `ChangeNotifierProvider`. In fact, you could have a stream in your model object and just call `notifyListeners()`. You wouldn't need a `StreamProvider` at all in that case.
- You could use a `StreamProvider` to implement the BLoC pattern.

ValueListenableProvider

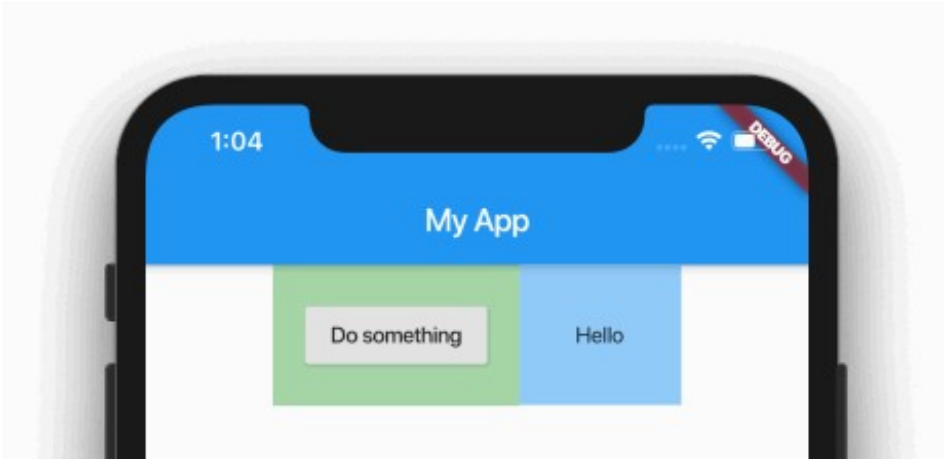
Feel free to scroll past this one. It's kind of like `ChangeNotifierProvider` . . . but more complicated . . . and without any obvious added value.

If you have a `ValueNotifier` like this,

```
class MyModel {  ValueNotifier<String> someValue = ValueNotifier('Hello');  void
doSomething() {
   someValue.value = 'Goodbye';
 }
}
```

then you can listen to any changes in it with `ValueListenableProvider`. However, if you want to call a method on the model from the UI, then you also need to provide the model. Thus, in the following code you can see a `Provider` provides `MyModel` to a `Consumer` that gives the `ValueNotifier` in `MyModel` to the `ValueListenableProvider`.

Here is the full code:



Notes:

- Pressing the “Do something” button makes “Hello” change to “Goodbye” because of the `ValueListenableProvider`.
- It would probably be better to use `Provider.of<MyModel>(context, listen: false)` rather than a `Consumer` at the top of the widget tree. Otherwise we are rebuilding the whole tree every time.
- `Provider<MyModel>` gives `myModel` to both the `ValueListenableProvider` and to the “Do something” button closure.
- The `Consumer<String>` for the `Text` widget knew to get its value from the `ValueListenableProvider<String>` because the `String` types matched.
- Seriously, making this example was a pain, especially trying to insert a `Consumer` at the top of the widget tree and getting all the brackets and parentheses mixed up. Using a `MultiProvider` (see below) would have improved it, though. Or just save yourself some pain and use a `ChangeNotifierProvider`.

ListenableProvider

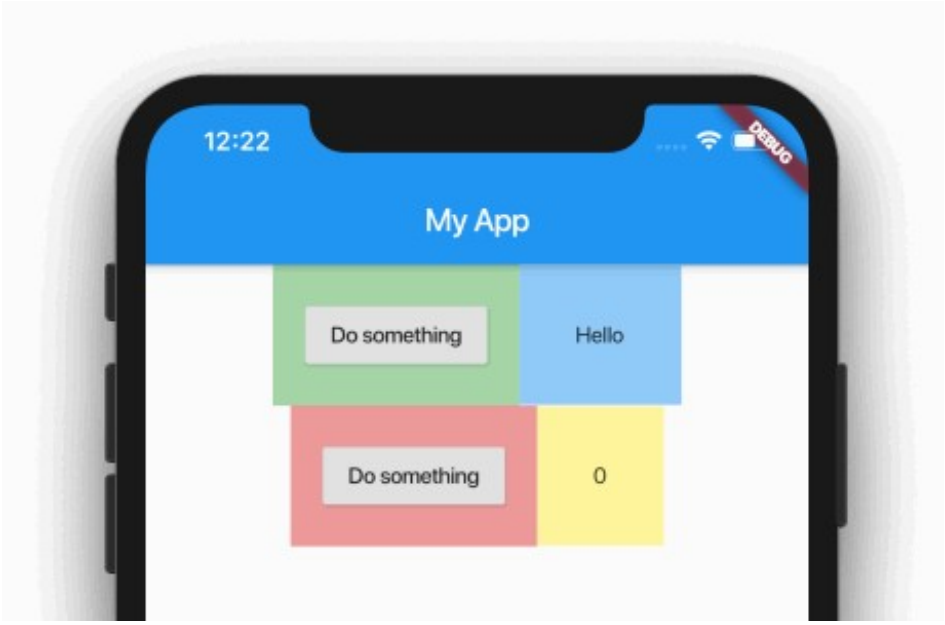
You would only use this if you need to build your own special provider. Even the documentation says that you probably want a `ChangeNotifierProvider` instead. So I am going to ignore `ListenableProvider` for now, though I may update this section in the future if I find a good use case.

MultiProvider

Up to now our examples have only used one model object. If you need to provide a second type of model object, you could nest the providers (similarly to what I did in the `ValueListenableProvider` example above). However, all that nesting is messy. A neater way to do it is to use a `MultiProvider`.

In the following example there are two different models that are provided with two `ChangeNotifierProviders`.

Here is the full code. It is a little long. Just scroll down noticing the `MultiProvider`, the `Consumers`, and the two model classes at the bottom:



Notes:

- Pressing the first “Do something” button will change the “Hello” to “Goodbye”. Pressing the second “Do something” button will change the “0” to “5”.
- There isn’t much different between this and the single `ChangeNotifierProvider`. The way that the different Consumers get the right model is by the type they indicate. That is, `Consumer<MyModel>` gets `MyModel`, and `Consumer<AnotherModel>` gets `AnotherModel`.

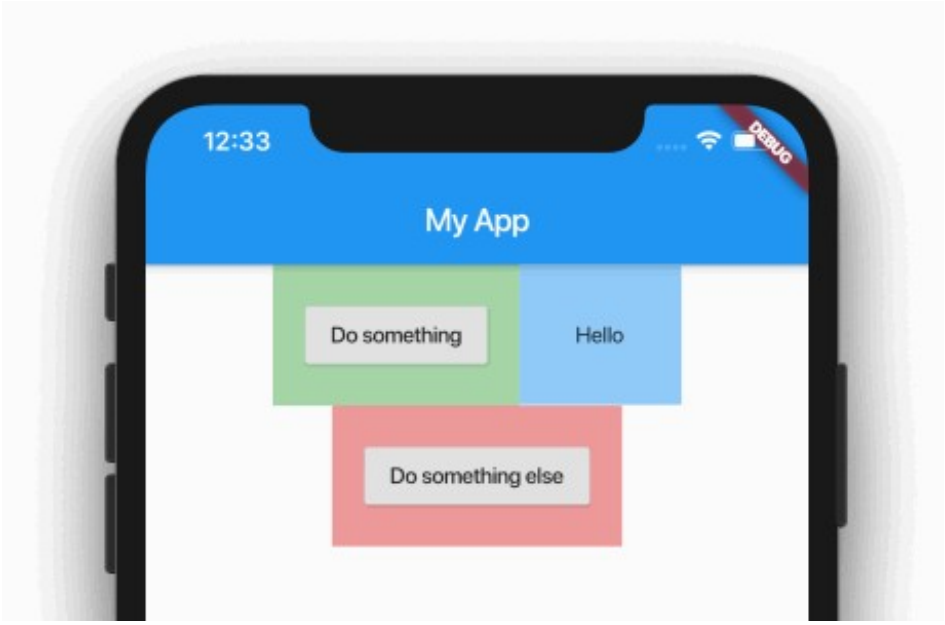
ProxyProvider

What if you have two models that you want to provide, but one of the models depends on the other one? In that case you can use a `ProxyProvider`. A `ProxyProvider` takes the value from one provider and lets it be injected into another provider.

The way you set up a `ProxyProvider` can be confusing at first, so let me add a little explanation about that.

The basic `ProxyProvider` has two types in the angle brackets. The first type is what the second type depends on. That is, it is a model that has already been provided by another `Provider`. It gets injected into the second model type in the `update` closure. The third parameter (`anotherModel`) stores the previous built value, but we don’t use that here. We just pass `myModel` into the constructor of `AnotherModel`.

Here is the full code for the example:



Notes:

- The text starts of saying “Hello”.
- When you press the “Do something” button, this has `MyModel` change the text to “Goodbye”. `MyModel` notifies its listener (`ChangeNotifierProvider`) and the UI gets rebuilt with the new text.
- When you press the “Do something else” button, `AnotherModel` takes `MyModel` (that was injected by the `ProxyProvider`) and changes its text to “See you later”. Because `MyModel` notifies its listeners of changes, the UI again gets updated. If `AnotherModel` had its own data that got changed, the UI would not be updated because `ProxyProvider` does not listen for changes. You would need a `ChangeNotifierProxyProvider` for that.
- `ProxyProvider` was sufficiently confusing for me. `ChangeNotifierProxyProvider` has even more special caveats and warnings. For that reason I am not going to add an example for it at this time. You can check out the documentation, though.
- I am in agreement with FilledStacks that the GetIt package is an easier way to handle dependency injection than `ProxyProvider` .

Provider builder and value constructors

Before I conclude, I want to explain one more thing that was confusing as I was learning to use Provider.

Most (if not all) of the Provider widgets have two kinds of constructors. The basic constructor takes a `create` function in which you create your model object. We did that in most of the examples above.

```
Provider<MyModel>(
  create: (context) => MyModel(),
  child: ...
)
```

You can see that the `MyModel` object was created in the `create` function.

If your object has already been created and you just want to provide a reference to it, then you can use the named constructor called `value`:

```
final myModel = MyModel();...Provider<MyModel>.value(
   value: myModel,
   child: ...
)
```

Here `MyModel` was previously created and was just passed in as a reference. You would do this if you had initialized your model in the `initState()` method so that you could call a method on the model to load data from the network.

Conclusion

After all this, my advice is that you can ignore most of the classes in the Provider package. Just learn how to use `ChangeNotifierProvider` and `Consumer`. Every now and then you might use a `Provider` widget if you don't need to update the UI. The logic for Futures and Streams can all be put into your model class that notifies `ChangeNotifierProvider`. No need for `FutureProvider` or `StreamProvider`. Most of the time you won't need a `MultiProvider` either if you have a view model for each screen or page. And for injecting dependencies in your view models, GetIt will handle that. No need to worry about `ProxyProvider`. This post gives some very specific help with what I have described here.

Even though I am telling you to ignore most of the Provider package, I do like it at its core. When you just focus the `ChangeNotifierProvider`, it really is an easy way to handle app state and architecture.

Update

I'm starting to have a change of heart about `ValueListenableProvider`. You can use it with an immutable state model object, and immutability has some advantages. See this video also about `ValueNotifier`.

Thanks

I've read a lot of different explanations and tutorials about Provider, but special thanks for the inspiration from these sources:

Flutter Community

4K