How to Use the Provider Pattern in Flutter

freecodecamp.org/news/provider-pattern-in-flutter/

June 12, 2020

12 June 2020 / #Flutter

Ayusch Jain

In this post we'll take a look at the provider pattern in Flutter. Some other patterns, such as BLoC Architecture, use the provider pattern internally. But the provider pattern is far easier to learn and has much less boilerplate code.

In this post, we'll take the default Counter app provided by Flutter and refactor it to use the provider pattern.

If you want to know what the Flutter team at Google has to say about the provider pattern, check out this 2019 talk.

If you want to learn more about BLoC Architecture, check it out here.

Getting Started

Create a new Flutter project and name it whatever you want.

First we need to remove all the comments so that we have a clean slate to work with:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
```

```
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Now add the dependency for the provider pattern in the `pubspec.yaml` file. At the time of writing, the latest version is 4.1.2.

Here's how your `pubspec.yaml` file will look now:

```
name: provider_pattern_explained
description: A new Flutter project.

publish_to: 'none'

version: 1.0.0+1

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  provider: ^4.1.2

  cupertino_icons: ^0.1.3

dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  uses-material-design: true
```

The default app is basically a stateful widget which rebuilds every time you click the `FloatingActionButton` (which calls `setState()`).

But now we're going to convert it into a stateless widget.

Creating the Provider

Let's go ahead and create our provider. This will be the single source of truth for our app. This is where we'll store our state, which in this case is the current count.

Create a class named `Counter` and add the `count` variable:

```
import 'package:flutter/material.dart';

class Counter {
  var _count = 0;
}
```

To convert it into a provider class, extend `ChangeNotifier` from the `material.dart` package. This provides us with the `notifyListeners()` method, and will notify all the listeners whenever we change a value.

Now add a method to increment the counter:

```
import 'package:flutter/material.dart';

class Counter extends ChangeNotifier {
  var _count = 0;
  void incrementCounter() {
    _count += 1;
  }
}
```

At the end of this method we'll call `notifyListeners()`. This will trigger a change all over the app to whichever widget is listening to it.

That's the beauty of the provider pattern in Flutter – you don't have to care about manually dispatching to streams.

Finally, create a getter to return the counter value. We'll use this to display the latest value:

```
import 'package:flutter/material.dart';

class Counter extends ChangeNotifier {
  var _count = 0;
  int get getCounter {
    return _count;
  }

  void incrementCounter() {
    _count += 1;
    notifyListeners();
  }
}
```

Listening to button clicks

Now that we have the provider set up, we can go ahead and use it in our main widget.

First, let's convert `MyHomePage` to a stateless widget instead of a stateful one. We'll have to remove the `setState()` call since that's only available in a `StatefulWidget`:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatelessWidget {
  int _counter = 0;
  final String title;
  MyHomePage({this.title});
  void _incrementCounter() {}
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

With this done, we can now use the provider pattern in Flutter to set and get the counter value. On each button click we need to increment the counter value by 1.

So, in the `_incrementCounter` method (which is called when the button is pressed) add this line:

```
Provider.of<Counter>(context, listen: false).incrementCounter();
```

What's happening here is that you've asked Flutter to go up in the *widget tree* and find the first place where `Counter` is provided. (I'll tell you how to provide it in the next section.) This is what `Provider.of()` does.

The generics (values inside <> brackets) tell Flutter what type of provider to look for. Then Flutter goes up through the widget tree until it finds the provided value. If the value isn't provided anywhere then an exception is thrown.

Finally, once you've got the provider, you can call any method on it. Here we call our `incrementCounter` method.

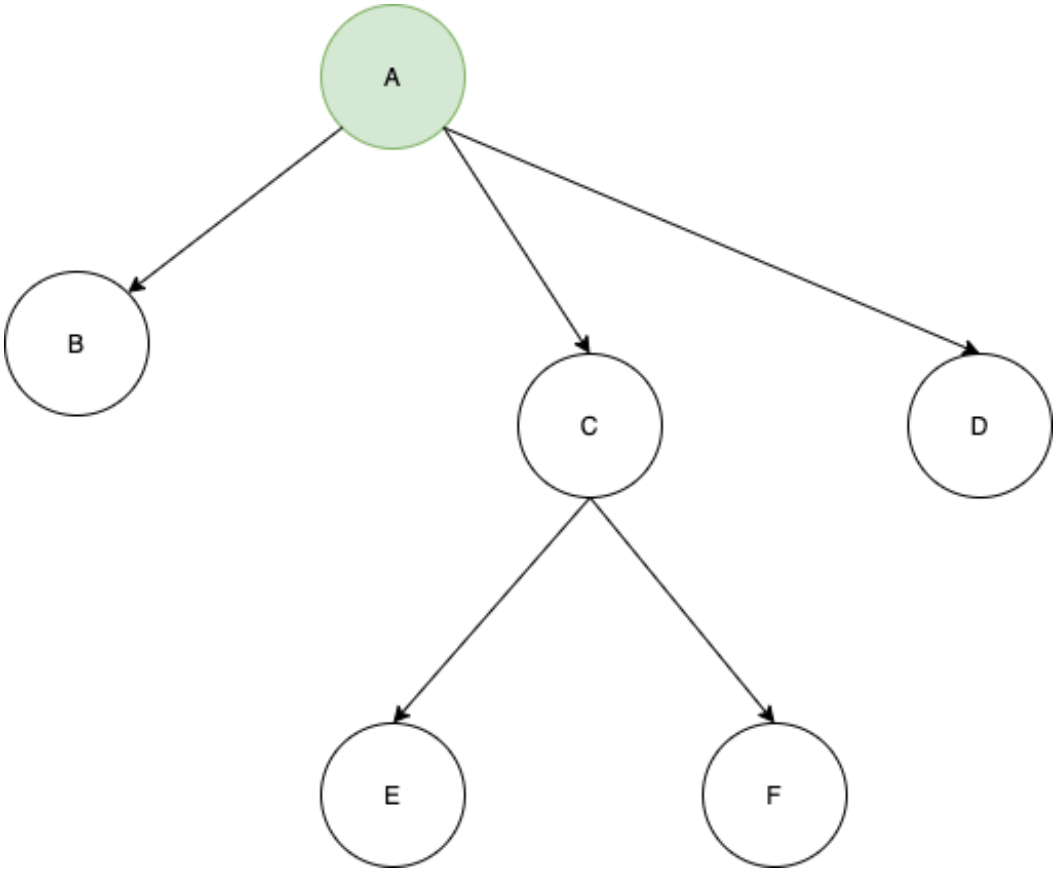
But we also need a context, so we accept the context as an argument and alter the `onPressed` method to pass the context as well:

```
void _incrementCounter(BuildContext context) {
  Provider.of<Counter>(context, listen: false).incrementCounter();
}
```

Note: We've set listen to false because we don't need to listen to any values here. We're just dispatching an action to be performed.

Providing the Provider

The provider pattern in Flutter will look for the latest value provided. The diagram below will help you better understand.

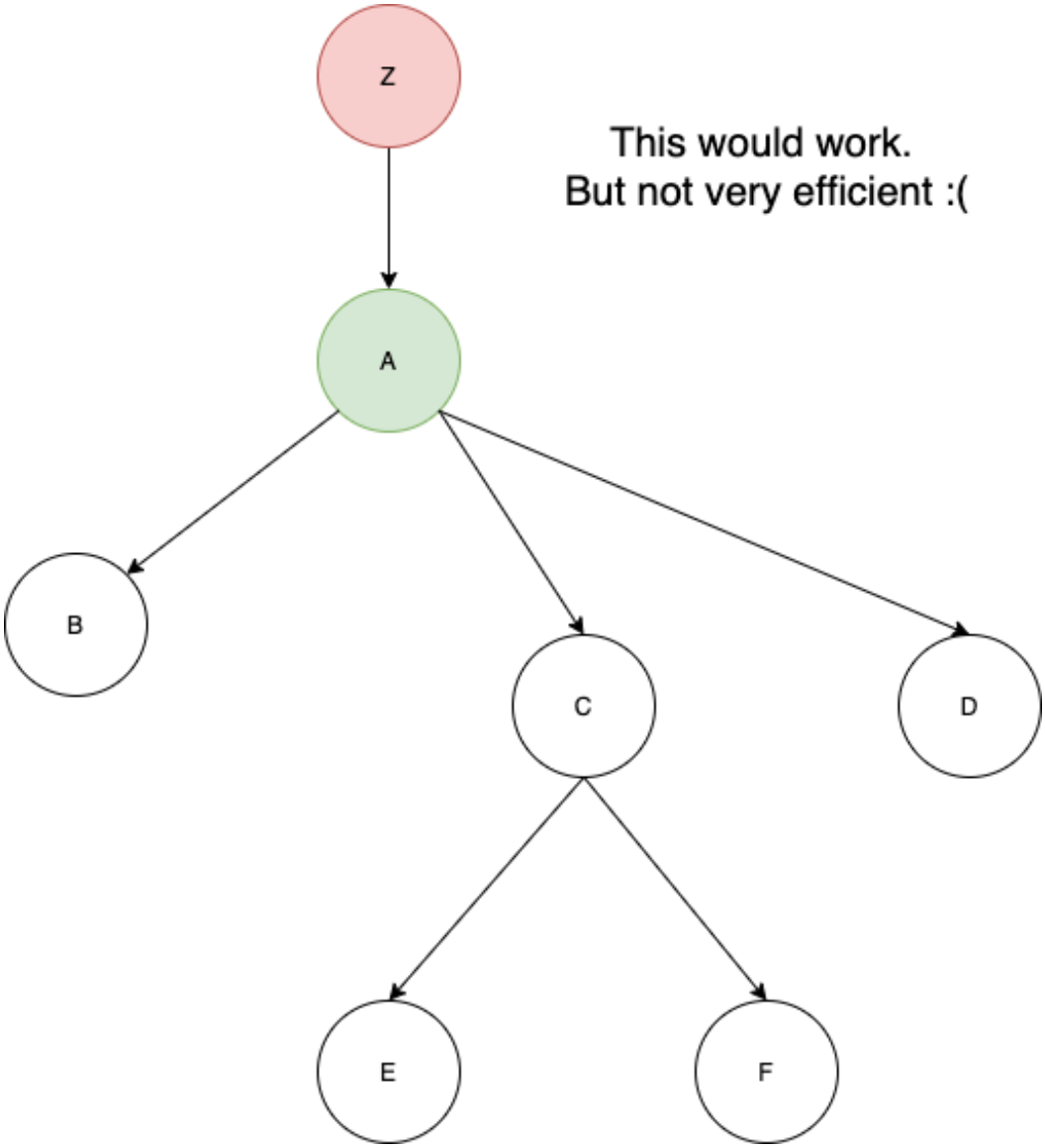


In this diagram the **GREEN** object **A** will be available to the rest of the elements below it, that is **B, C, D, E,** and **F.**

Now suppose we want to add some functionality to the app and we create another provider, **Z.** Z is required by E and F.

So where is the best place to add that?

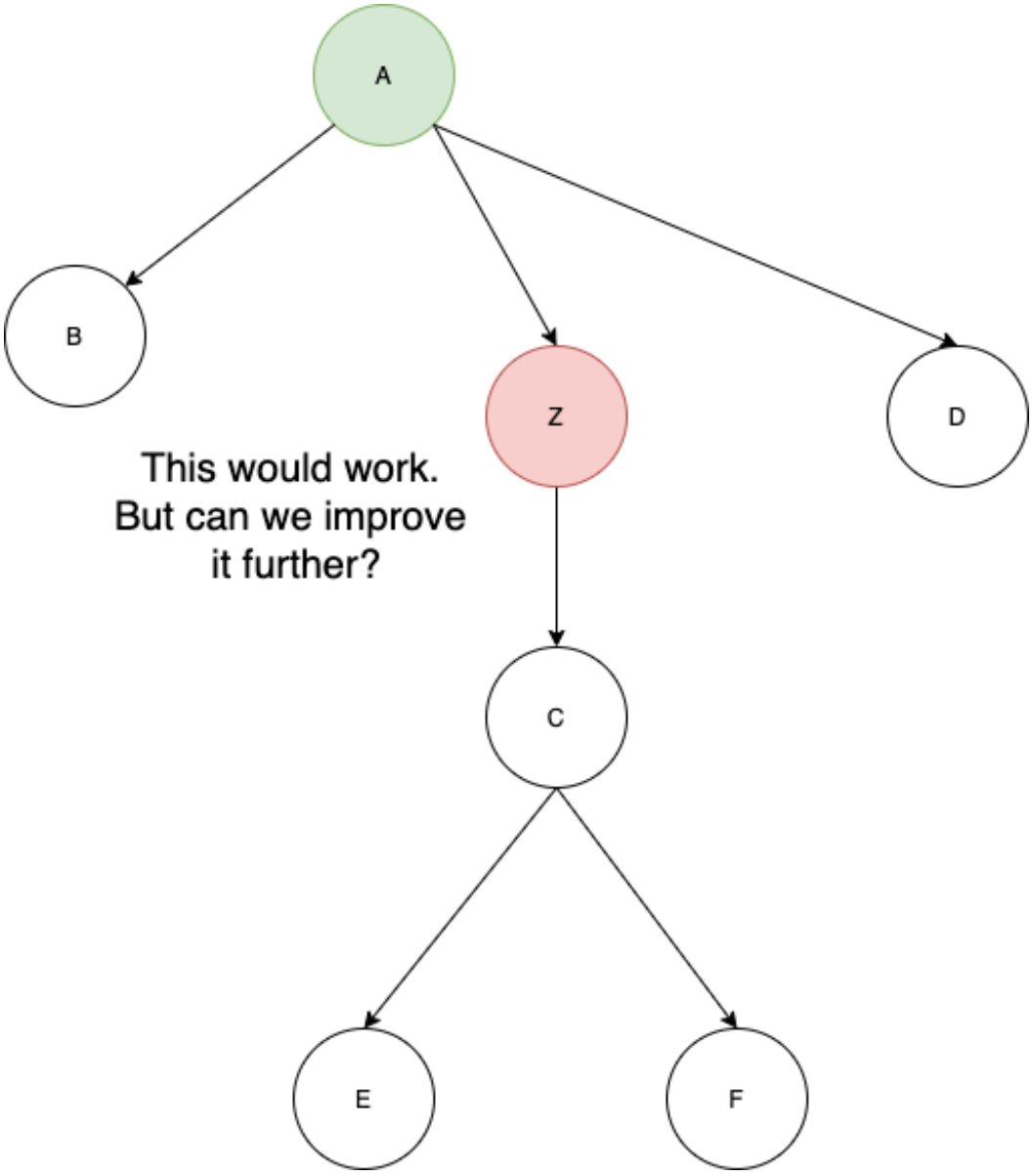
We can add it to the root above **A**. This would work:



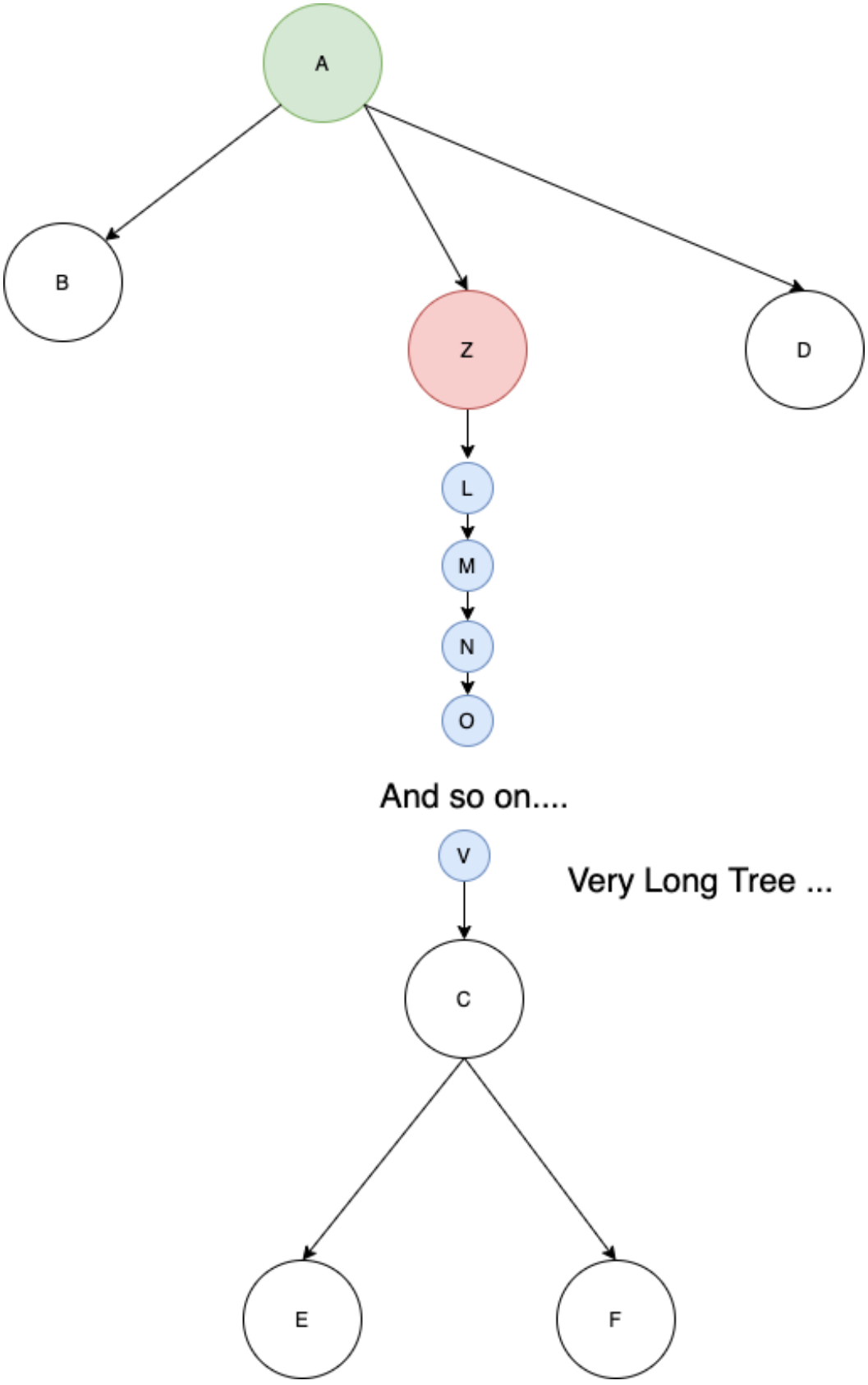
But this method is not very efficient.

Flutter will go through all the widgets above and then finally go to the root. If you have very long widget trees – which you definitely will in a production app – then it's not a good idea to put everything at the root.

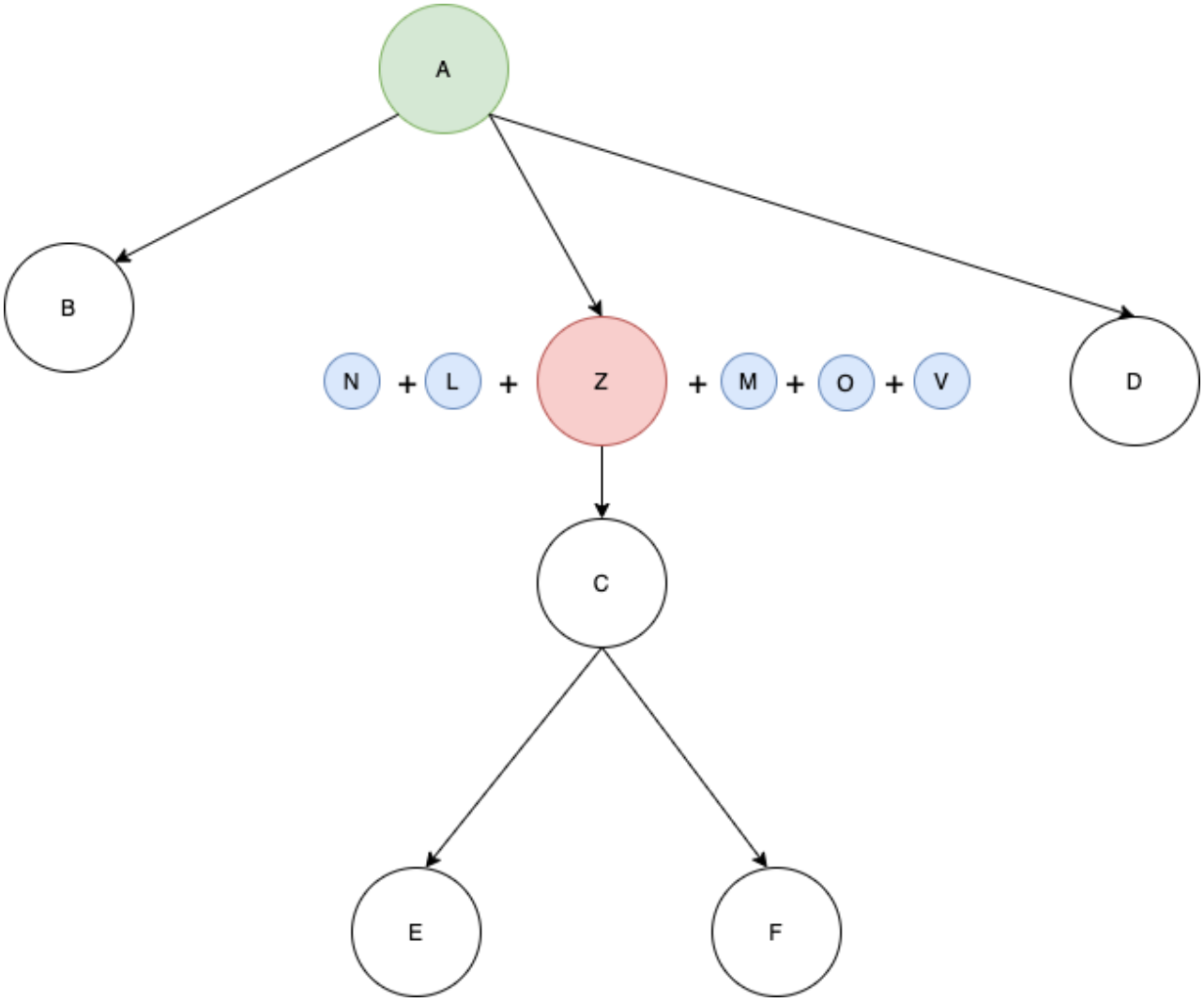
Instead, we can look at the common denominator of E and F. That is C. So if we put Z just above E and F, it would work.



But what if we want to add another object **X** that'srequired by E and F? We'll do the same thing. But notice how the tree keeps growing.



There’s a better way to manage that. What if we provide all the objects at one level?



This is perfect, and is how we'll eventually implement our provider pattern in Flutter. We'll make use of something called `MultiProvider` which lets us declare multiple providers at one level.

We'll get `MultiProvider` to wrap the `MaterialApp` widget:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider.value(
          value: Counter(),
        ),
      ],
      child: MaterialApp(
        title: 'Flutter Demo',
        theme: ThemeData(
          primarySwatch: Colors.blue,
          visualDensity: VisualDensity.adaptivePlatformDensity,
        ),
        home: MyHomePage(title: "AndroidVille Provider Pattern"),
      ),
    );
  }
}
```

With this, we've provided the provider to our widget tree and can use it anywhere below this level in the tree.

There's just one more thing left: we need to update the value that's displayed.

Updating the text

To update the text, get the provider in the build function of your `MyHomePage` widget. We'll use the getter we created to get the latest value.

Then just add this value to the text widget below.

And we're done! This is how your final `main.dart` file should look:

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'package:provider_pattern_explained/counter.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider.value(
          value: Counter(),
        ),
      ],
      child: MaterialApp(
```

```
        title: 'Flutter Demo',
        theme: ThemeData(
          primarySwatch: Colors.blue,
          visualDensity: VisualDensity.adaptivePlatformDensity,
        ),
        home: MyHomePage(title: "AndroidVille Provider Pattern"),
      ),
    );
  }
}

class MyHomePage extends StatelessWidget {
  final String title;
  MyHomePage({this.title});
  void _incrementCounter(BuildContext context) {
    Provider.of<Counter>(context, listen: false).incrementCounter();
  }

  @override
  Widget build(BuildContext context) {
    var counter = Provider.of<Counter>(context).getCounter;
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _incrementCounter(context),
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Note: We haven't set `listen:false` in this case because we want to listen to any updates in the count value.

Here's the source code on GitHub if you want to have a look: https://github.com/Ayusch/Flutter-Provider-Pattern.

Let me know if you have any issues.

Welcome to AndroidVille :)

AndroidVille is a community of Mobile Developers where we share knowledge related to Android Development, Flutter Development, React Native Tutorials, Java, Kotlin and much more.

Click on this link to join the AndroidVille SLACK workspace. It's absolutely free!

If you liked this article, feel free to share it on Facebook or LinkedIn. You can follow me on LinkedIn, Twitter, Quora, and Medium where I answer questions related to mobile development, Android, and Flutter.

Ayusch Jain

I'm an Android Engineer with a passion for developing apps for mobile.

If you read this far, tweet to the author to show them you care. Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started