29. State & Stateful Widget Approach

Introduction

This is the most obvious approach and uses Flutter Widgets in the most obvious manner possible.

The purpose of this chapter is to learn this approach and its shortcomings.

Approach

- Store state in Stateful Widgets at a high-enough level in the Widget tree to ensure that the data is not repeated.
- Pass state from parent Widgets to child Widgets through the constructor.
- Pass event handler method (that modifies state) from parent Widget methods to child Widgets through the constructor. Child Widgets can then invoke method to change state in Parent Widget.

Exercise – ‘state_and_stateful_widget’

Introduction

We start off by creating a create basic app with Stateful and Stateless Widgets.

Later on, we add some state & event handling so that the user can select a car and see it highlighted.

- The car selection comes from a tap event in the lower-level CarWidget.
- It changes the selected car state in the higher-level MyHomePageWidget.

Step 1 – Create Default Flutter App

Follow the instructions in Generate Your First App
Leave project open.

Step 2 – Replace Application Code

Replace contents of file ‘main.dart’ in folder ‘lib’ with the following:

```
import ‘package:flutter/material.dart’;

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
  return new MaterialApp(
    title: ‘Flutter Demo’,
    theme: new ThemeData(
      primarySwatch: Colors.blue,
```

```
    ),
    home: new MyHomePage(),
  );
}
}

class Car {
String _make;
String _model;
String _imageSrc;

Car(this._make, this._model, this._imageSrc);

operator ==(other) =>
    (other is Car) && (_make == other._make) && (_model == other._model);

int get hashCode => _make.hashCode ^ _model.hashCode ^ _imageSrc.hashCode;

}

class MyHomePage extends StatefulWidget {
@override
_HomePageState createState() => _HomePageState("Cars");
}

class _HomePageState extends State<MyHomePage> {
String _title;
```

```
List<Car> _cars;

_HomePageState(this._title) {
  _cars = [
    Car(
      “Bmw”,
      “M3”,
      “Https://media.ed.edmunds-media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_o
Https://media.ed.edmunds-media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_o
“,
    ),
    Car(
      “Nissan”,
      “GTR”,
      “Https://media.ed.edmunds-media.com/nissan/gt-r/2018/oem/2018_nissan_gt-r_coupe_nismo_fq_oem_1_150.jpg
Https://media.ed.edmunds-media.com/nissan/gt-r/2018/oem/2018_nissan_gt-r_coupe_nismo_fq_oem_1_150.jpg
”,
    ),
    Car(
      “Nissan”,
      “Sentra”,
      “Https://media.ed.edmunds-
```

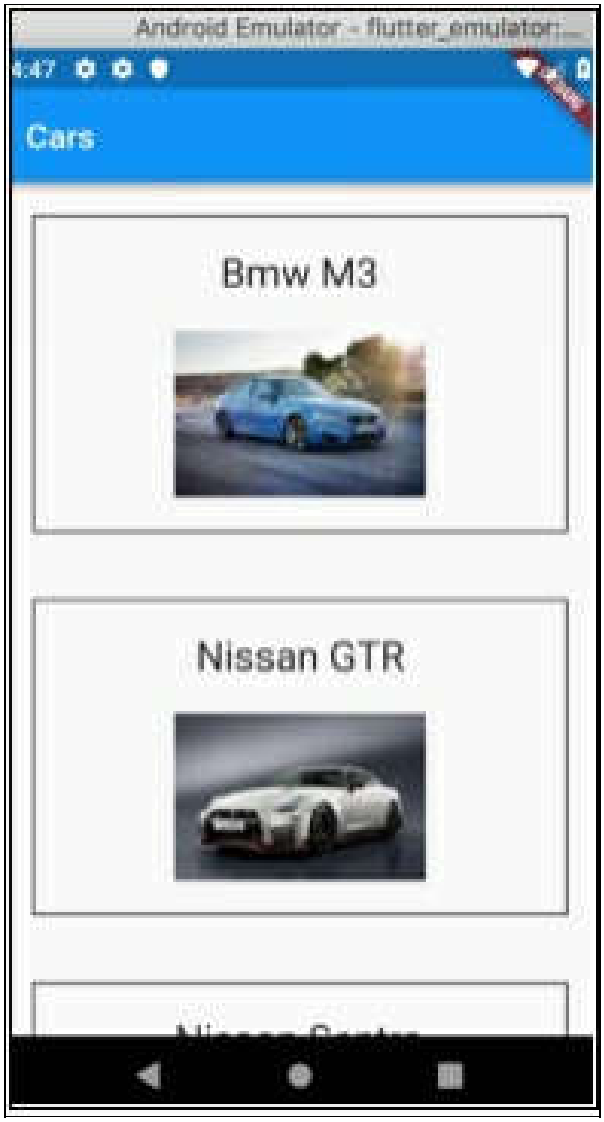
```
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg
Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg
",
    )
  ];
}

@override
Widget build(BuildContext context) {
  List<CarWidget> carWidgets = _cars.map((Car car) {
    return CarWidget(car);
  }).toList();
  return new Scaffold(
      appBar: new AppBar(
        title: new Text(_title),
      ),
      body: new ListView(children: carWidgets));
}
}

class CarWidget extends StatelessWidget {
CarWidget(this._car) : super();

final Car _car;
```

```
@override
Widget build(BuildContext context) {
  return Padding(
    padding: EdgeInsets.all(20.0),
    child: Container(
      decoration: BoxDecoration(border: Border.all()),
      padding: EdgeInsets.all(20.0),
      child: Center(
        child: Column(children: <Widget>[
        Text('${_car._make} ${_car._model}',
          style: TextStyle(fontSize: 24.0)),
        Padding(
          padding: EdgeInsets.only(top: 20.0),
          child: Image.network(_car._imageSrc))
      ])));
}
}
```

Step 3 – Open Emulator & Run

Follow the instructions in Open Android Emulator & Run Your First App

You should get something like the following as it is somewhat similar to the previous example:



Summary

- The MyApp & Material App Widgets are unchanged.
- We declare a new class called Car.
    - This will store information about each car: its make, model and image.

    - Note that the '==' operator is overloaded so it considers two Cars equal if they have the same make and model.
- The MyHomePage Stateless Widget has become two different widgets instead:
    - MyHomePage StatefulWidget
    - MyHomePageState State Object
        - This holds the App Bar title and the list of Car objects. These are initiated in the constructor.
        - The State object contains the 'build' method that converts the list of Car objects into a list of CarWidgets. Then it returns a Scaffold containing the AppBar and a ListView containing the list of CarWidgets.
    - CarWidget
        - This displays a car's make, model and image.
        - Notice that it now accepts a Car object in the constructor. This gives it all the info to display a car's make, model and image.

Step 4– Add Car Selection

This is going to be achieved by holding state in the MyHomePage state object.
This state is going to be set by a method. This method is going to be passed to each Car Widget so it can be invoked by the Car Widget when the user taps on it.

Modify MyHomePageState

- We add variable '_selectedCar' to store which car is selected.
- We add a method '_selectionHandler' to handle car selection.
    - This provides an inline JavaScript function that sets the variables '_title' and '_selectedCar'.
    - This inline JavaScript function is passed to setState. Using 'setState' tells Flutter that the state of this object has changed and that this Widget will need to be re-rendered.
- We change the code that constructs the CarWidgets to include 2 additional constructor arguments:
    - A boolean indicating if the car is the selected car.

- The selection handler method that handles the car selection in this class.

```
class MyHomePageState extends State<MyHomePage> {
String _title;
List<Car> _cars;
 Car _selectedCar;

MyHomePageState(this._title) {
  _cars = [
    Car(
      “Bmw”,
      “M3",
      “Https://media.ed.edmunds-
media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_o
    ),
    Car(
      “Nissan”,
      “GTR”,
      “Https://media.ed.edmunds-media.com/nissan/gt-
r/2018/oem/2018_nissan_gt-
r_coupe_nismo_fq_oem_1_150.jpg”,
    ),
    Car(
      “Nissan”,
      “Sentra”,
      “Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
```

```
turbo_fq_oem_4_150.jpg",
    )
  ];
}

 void _selectionHandler(Car selectedCar) {
  setState(() {
    _title = 'Selected ${selectedCar._make}
${selectedCar._model}';
    _selectedCar = selectedCar;
  });
}

@override
Widget build(BuildContext context) {
  List<CarWidget> carWidgets = _cars.map((Car car) {
    return CarWidget(car, car == _selectedCar,
_selectionHandler);
  }).toList();
  return new Scaffold(
      appBar: new AppBar(
        title: new Text(_title),
      ),
      body: new ListView(children: carWidgets));
}
}
```

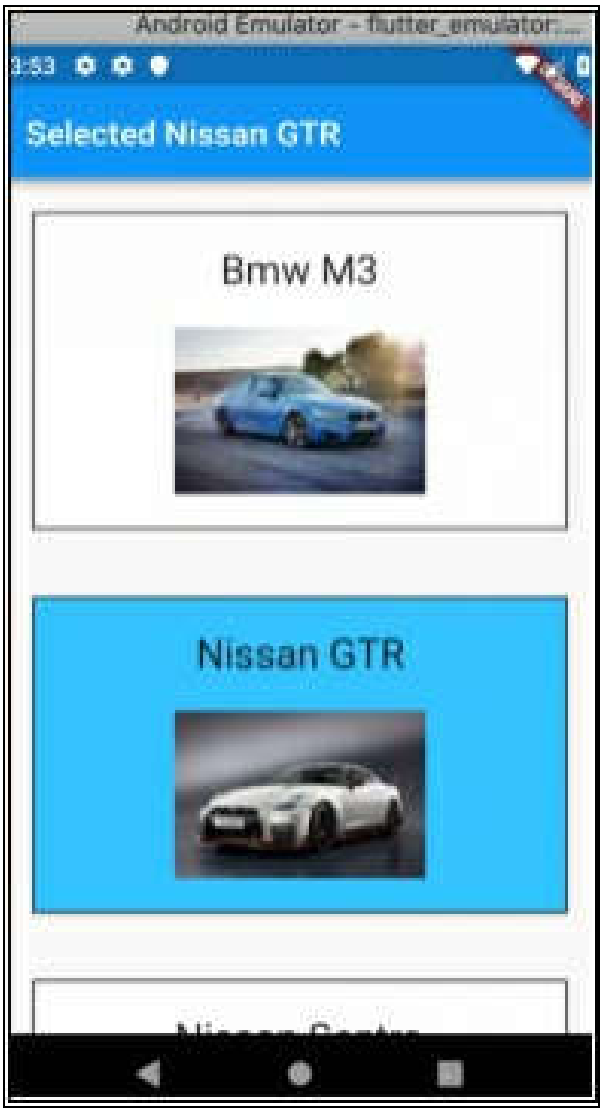
Modify CarWidget

- We add instance variable '_isSelected' to store if this car is selected or not.
- We add instance variable '_parentSelectionHandler' to store the selection handler method from the parent MyHomePageState class.
- We modify the constructor to accept & set these two instance variables.
- We add a new method '_handleTap' to handle the 'onTap' event from the GestureDetector. This method invokes the '_parentSelectionHandler' from the parent MyHomePageState class.
- We modify the 'build' method.
    - We wrap the Container with a GestureDetector. This is so we can listen for the 'onTap' event.
    - We modify the 'BoxDecoration' to set the background color according to if the instance variable 'isSelected' is set to true or false. If true the background color is set to blue, otherwise white.

```
class CarWidget extends StatelessWidget {

CarWidget(this._car, this._isSelected,
this._parentSelectionHandler)

    : super();
```

```
final Car _car;
 final bool _isSelected;
final ValueChanged<Car> _parentSelectionHandler;

void _handleTap() {
  _parentSelectionHandler(_car);
}

@override
Widget build(BuildContext context) {
   return Padding(
       padding: EdgeInsets.all(20.0),
       child: GestureDetector(
        onTap: _handleTap,
           child: Container(
               decoration: BoxDecoration(
                   color: _isSelected ? Colors.blue : Colors.white,
                   border: Border.all()),
               padding: EdgeInsets.all(20.0),
               child: Center(
                   child: Column(children: <Widget>[
                 Text('${_car._make} ${_car._model}',
                     style: TextStyle(fontSize: 24.0)),
                 Padding(
                     padding: EdgeInsets.only(top: 20.0),
                     child: Image.network(_car._imageSrc))
               ]))));
```

```
}
}
```



Further Reading

- Adding Interactivity to Your Flutter App: Https://flutter.io/docs/development/ui/interactive
- Pete Hunt at Facebook wrote a superb article

here.

The article may be about React but many of the same rules apply.
Https://facebook.github.io/react/docs/thinking-in-react.html.

30. State & InheritedWidget Approach

Introduction

This is a way to access State that is stored in a higher-level Widget (called an InheritedWidget) from a lower-level Widget. Think of it like this: “Reach Up the Tree and Get Data”. Flutter uses InheritedWidgets itself. The Theme Widget is in an InheritedWidget.

The purpose of this chapter is to learn what InheritedWidgets are and how to use them.

Approach

- This approach removes many of the requirements to use Stateful Widgets, often enabling the user to use Stateless Widgets instead.
- You create a ‘state holder’ class that acts as a Widget in the Widget hierarchy. This class extends InheritedWidget, stores the state data and has a single child widget.
- All the Widgets below this class can then be Stateless Widgets and they can use the BuildContext to access this InheritedWidget and its state data.
- To see an example of this, see Chapter ‘State &

InheritedWidget Approach’.

Exercise – ‘state_and_inherited_widget_add’

In this exercise, I put the state for the car list into CarsInheritedWidget and I access it in CarWidget. I add a toolbar button to add another car to the list.

Please read the summary before starting this exercise. This exercise shows how using a state and an inherited Widget won’t work as expected.

Step 1 – Create Default Flutter App

Follow the instructions in Generate Your First App
Leave project open.

Step 2 – Replace Application Code

Replace contents of file ‘main.dart’ in folder ‘lib’ with the following:

```
import ‘package:collection/collection.dart’;
import ‘package:flutter/material.dart’;

void main() => runApp(new MyApp());

class Car {
String _make;
```

```
String _model;
String _imageSrc;

Car(this._make, this._model, this._imageSrc);

operator ==(other) =>
    (other is Car) && (_make == other._make) && (_model == other._model);

int get hashCode => _make.hashCode ^ _model.hashCode ^ _imageSrc.hashCode;
}

class CarsInheritedWidget extends InheritedWidget {
List<Car> _cars = [
  Car(
    “Bmw”,
    “M3",
    “Https://media.ed.edmunds-media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_c
Https://media.ed.edmunds-media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_c
“,
  ),
  Car(
    “Nissan”,
    “GTR”,
    “Https://media.ed.edmunds-media.com/nissan/gt-r/2018/oem/2018_nissan_gt-
```

```
r_coupe_nismo_fq_oem_1_150.jpg
Https://media.ed.edmunds-media.com/nissan/gt-r/2018/oem/2018_nissan_gt-r_coupe_nismo_fq_oem_1_150.jpg
",
  ),
  Car(
    "Nissan",
    "Sentra",
    "Https://media.ed.edmunds-media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg
Https://media.ed.edmunds-media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg
",
  )
];

CarsInheritedWidget(child) : super(child: child);

List<Car> get cars {
  return _cars;
}

void addNissanSentra() {
  _cars.add(Car(
    "Nissan",
    "Sentra",
```

```
    “Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg
Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg
”,
  ));
}

@override
bool updateShouldNotify(CarsInheritedWidget old) => true;

static CarsInheritedWidget of(BuildContext context) {
  return
(context.inheritFromWidgetOfExactType(CarsInheritedWidget))
}
}

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
  return new MaterialApp(
    title: ‘Flutter Demo’,
    theme: new ThemeData(
      // This is the theme of your application.
      //
      // Try running your application with “flutter run”. You’ll
```

```
see the
      // application has a blue toolbar. Then, without quitting the app, try
      // changing the primarySwatch below to Colors.green and then invoke
      // “hot reload” (press “r” in the console where you ran “flutter run”,
      // or press Run > Flutter Hot Reload in IntelliJ). Notice that the
      // counter didn’t reset back to zero; the application is not restarted.
      primarySwatch: Colors.blue,
    ),
    home: CarsInheritedWidget(MyHomePage(title: ‘Flutter Demo Home Page’)),
  );
}
}

class MyHomePage extends StatelessWidget {
MyHomePage({Key key, this.title}) : super(key: key);

final String title;

@override
Widget build(BuildContext context) {
  List<CarWidget> carWidgets =
      CarsInheritedWidget.of(context).cars.map((Car car) {
    return CarWidget(car);
```

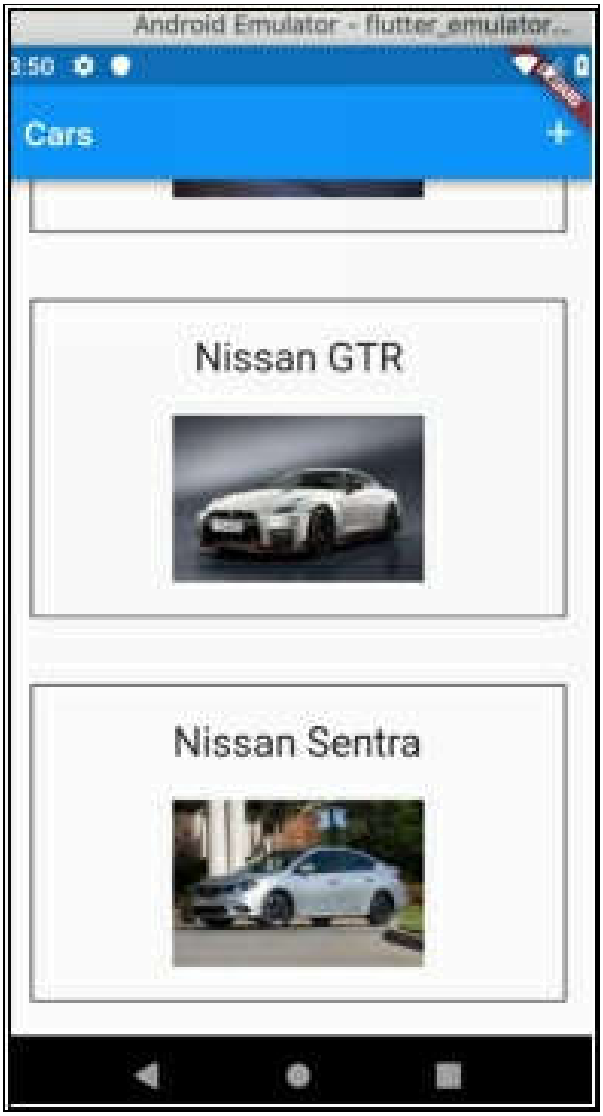
```
  }).toList();
  return new Scaffold(
      appBar: new AppBar(
        title: new Text("Cars"),
        actions: <Widget>[
          IconButton(
              icon: Icon(Icons.add),
              onPressed: () {
                CarsInheritedWidget.of(context).addNissanSentra();
              })
        ],
      ),
      body: new ListView(children: carWidgets));
}
}

class CarWidget extends StatelessWidget {
CarWidget(this._car) : super();

final Car _car;

@override
Widget build(BuildContext context) {
  return Padding(
      padding: EdgeInsets.all(20.0),
      child: Container(
          decoration: BoxDecoration(border: Border.all()),
```

```
        padding: EdgeInsets.all(20.0),
        child: Center(
          child: Column(children: <Widget>[
          Text('${_car._make} ${_car._model}',
            style: TextStyle(fontSize: 24.0)),
          Padding(
            padding: EdgeInsets.only(top: 20.0),
            child: Image.network(_car._imageSrc))
        ])));
}
}
```

Step 3 – Open Emulator & Run

Follow the instructions in Open Android Emulator & Run Your First App

You should get something like the following as it is somewhat similar to the previous example:

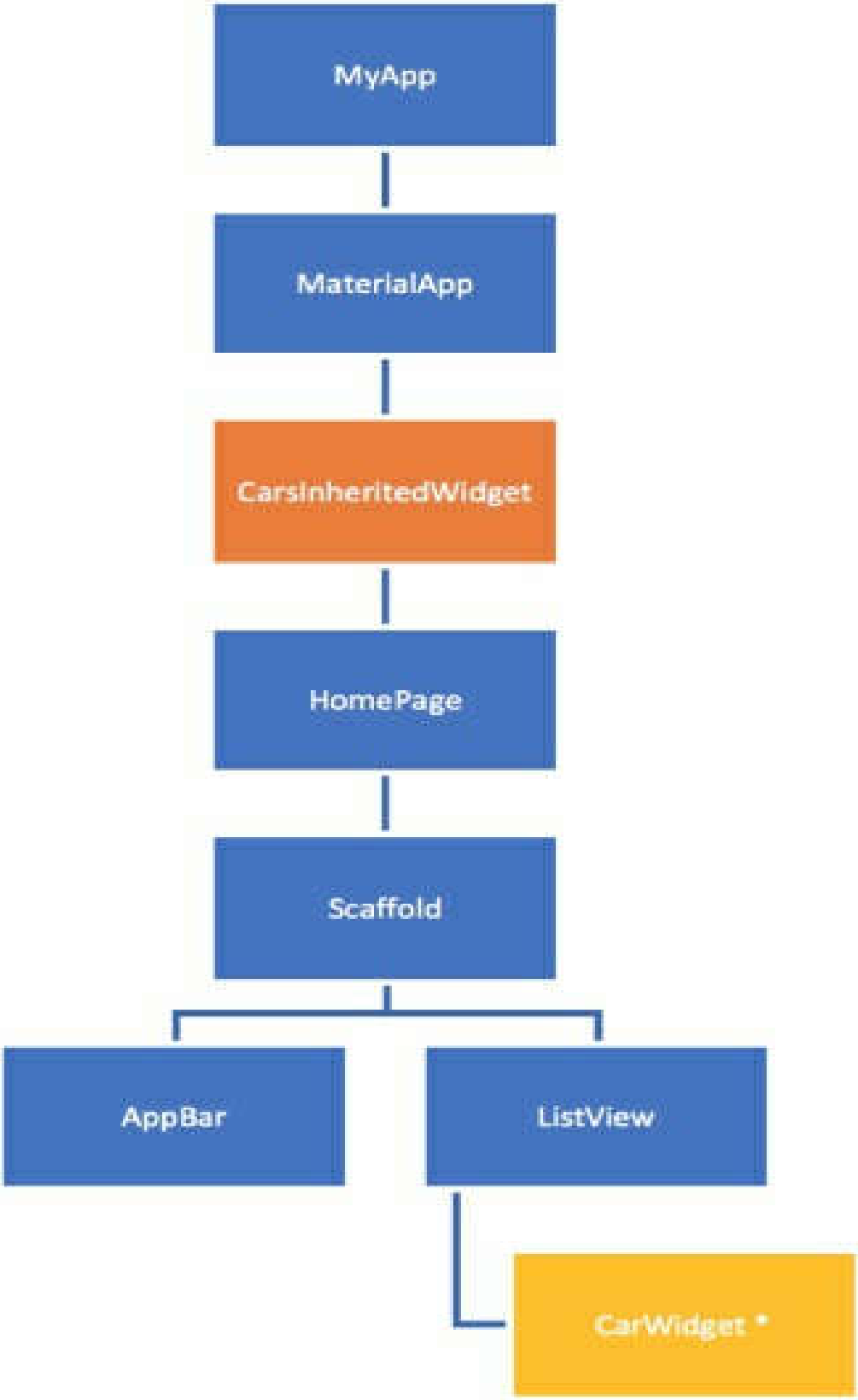


However, note that the Add button on the toolbar does not work!!!!!

Summary

We created a class CarsInheritedWidget that inherits from Inherited Object and we added into the Widget Tree, wrapping the HomePage Widget.

It seems we can access the state in that Widget, the list of Cars.
However, when we add a car it doesn’t show up.


MyApp
MaterialApp
CarsInheritedWidget
HomePage
Scaffold
AppBar
ListView
CarWidget *

After reading some articles, it became obvious that to mutate the State of an InheritedWidget and have the UI re-render the state changes**, you need to wrap the InheritedWidget in a StatefulWidget.**

This example is based on the article below:
Https://medium.com/flutter-io/managing-flutter-application-state-with-inheritedwidgets-1140452befe1

Exercise – ‘state_and_inherited_widget’

In this exercise, we get the State mutation to work on-screen and explain the changes.

Step 1 – Create Default Flutter App

Follow the instructions in Generate Your First App
Leave project open.

Step 2 – Replace Application Code

Replace contents of file ‘main.dart’ in folder ‘lib’ with the following:

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

void main() => runApp(new MyApp());
```

```
class Car {
  final String _make;
  final String _model;
  final String _imageSrc;

  const Car(this._make, this._model, this._imageSrc);

  operator ==(other) =>
      (other is Car) && (_make == other._make) && (_model
== other._model);

  int get hashCode => _make.hashCode ^ _model.hashCode ^
_imageSrc.hashCode;
}

class CarModel {
  const CarModel(this.carList);

  final List<Car> carList;

  @override
  bool operator ==(Object other) {
    if (identical(this, other)) {
      return true;
    } else if (other.runtimeType != runtimeType) {
      return false;
    } else {
      final CarModel otherModel = other;
      return IterableEquality().equals(otherModel.carList,
```

```
carList);
    }
  }

  int get hashCode => carList.hashCode;
}

class _ModelBindingScope<T> extends InheritedWidget {
  const _ModelBindingScope({Key key,
this.modelBindingState, Widget child})
      : super(key: key, child: child);

  final _ModelBindingState<T> modelBindingState;

  @override
  bool updateShouldNotify(_ModelBindingScope oldWidget)
=> true;
}

class ModelBinding<T> extends StatefulWidget {
  ModelBinding({Key key, @required this.initialModel,
this.child})
      : assert(initialModel != null),
        super(key: key);

  final T initialModel;
  final Widget child;

  _ModelBindingState<T> createState() =>
_ModelBindingState<T>();
```

```
  static Type _typeOf<T>() => T;

  static T of<T>(BuildContext context) {
    final Type scopeType = _typeOf<_ModelBindingScope<T>>
();
    final _ModelBindingScope<T> scope =
        context.inheritFromWidgetOfExactType(scopeType);
    return scope.modelBindingState.currentModel;
  }

  static void update<T>(BuildContext context, T newModel) {
    final Type scopeType = _typeOf<_ModelBindingScope<T>>
();
    final _ModelBindingScope<dynamic> scope =
        context.inheritFromWidgetOfExactType(scopeType);
    scope.modelBindingState.updateModel(newModel);
  }
}

class _ModelBindingState<T> extends
State<ModelBinding<T>> {
  T currentModel;

  @override
  void initState() {
    super.initState();
    currentModel = widget.initialModel;
  }
```

```
  void updateModel(T newModel) {
    if (newModel != currentModel) {
      setState(() {
        currentModel = newModel;
      });
    }
  }

  @override
  Widget build(BuildContext context) {
    return _ModelBindingScope<T>(
      modelBindingState: this,
      child: widget.child,
    );
  }
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Demo',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: ModelBinding<CarModel>(
```

```
        initialModel: const CarModel(const [
          Car(
            "Bmw",
            "M3",
            "Https://media.ed.edmunds-
media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_o
          ),
          Car(
            "Nissan",
            "GTR",
            "Https://media.ed.edmunds-media.com/nissan/gt-
r/2018/oem/2018_nissan_gt-
r_coupe_nismo_fq_oem_1_150.jpg",
          ),
          Car(
            "Nissan",
            "Sentra",
            "Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg",
          )
        ]),
        child: new MyHomePage(title: 'Flutter Demo Home
Page')),
    );
  }
}
```

```
class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  Widget build(BuildContext context) {
    CarModel model = ModelBinding.of(context);
    List<CarWidget> carWidgets = model.carList.map((Car car)
{
      return CarWidget(car);
    }).toList();
    return new Scaffold(
        appBar: new AppBar(
          title: new Text("Cars"),
          actions: <Widget>[
            IconButton(
                icon: Icon(Icons.add),
                onPressed: () {
                  List<Car> carList = List.from(model.carList);
                  carList.add(Car(
                    "Nissan",
                    "Sentra",
                    "Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg",
                  ));
                  ModelBinding.update(context, new
```

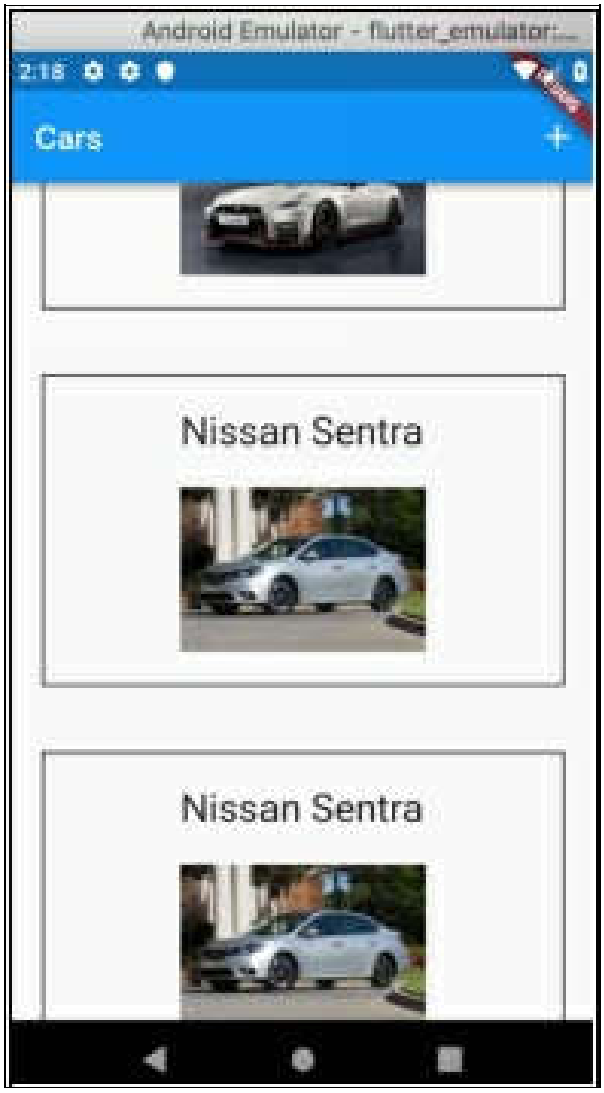
```
CarModel(carList));
              })
        ],
      ),
      body: new ListView(children: carWidgets));
  }
}

class CarWidget extends StatelessWidget {
  CarWidget(this._car) : super();

  final Car _car;

  @override
  Widget build(BuildContext context) {
    return Padding(
        padding: EdgeInsets.all(20.0),
        child: Container(
            decoration: BoxDecoration(border: Border.all()),
            padding: EdgeInsets.all(20.0),
            child: Center(
                child: Column(children: <Widget>[
              Text('${_car._make} ${_car._model}',
                  style: TextStyle(fontSize: 24.0)),
              Padding(
                  padding: EdgeInsets.only(top: 20.0),
                  child: Image.network(_car._imageSrc))
```

```
            ])));
  }
}
```

Step 3 – Open Emulator & Run

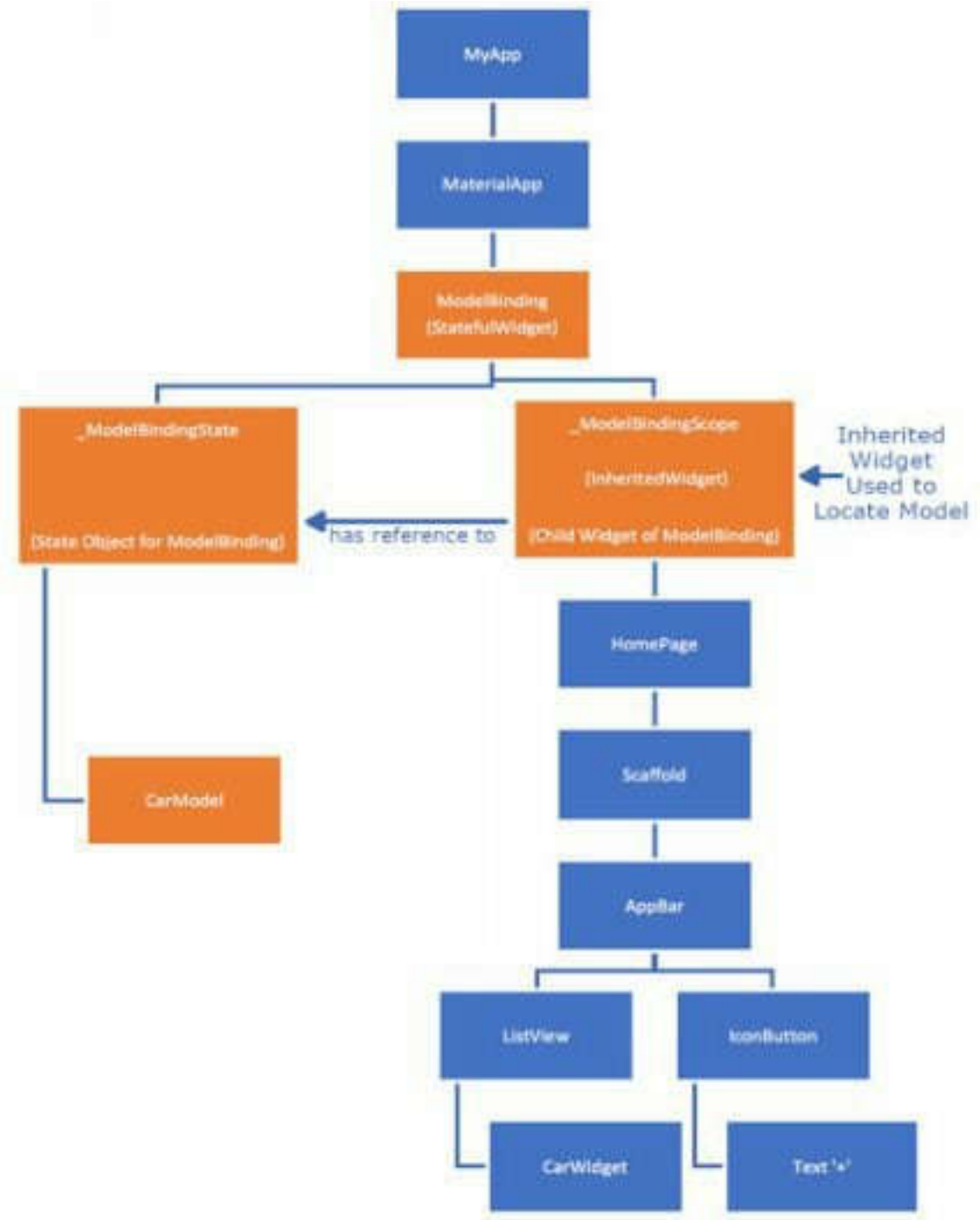
Follow the instructions in Open Android Emulator & Run Your First App

This works much better; the user interface responds to the ‘+’ button and adds another car to the list.



Summary

In this example, we to wrap the InheritedWidget in a StatefulWidget to enable it to re-render part of the Widget Tree.



- ModelBinding.
    - The ModelBinding class is a Stateful Widget. Remember, to mutate the State

of an InheritedWidget and have the UI re-render the state changes, you need to wrap the InheritedWidget in a StatefulWidget. This is what the ModelBinding class does. The ‘update’ method is used to update the state (the CarModel) in this Stateful Widget.

- _ModelBindingState.
    - This is the State for the ModelBinding Stateful Widget. It contains the CarModel. The ‘updateModel’ method is used to replace the model (the CarModel) in this class with a new one, calling ‘setState’ to force the UI to re-render the state changes.

- _ModelBindingScope.
    - This is an InheritedWidget, used to locate items in the Widget Tree. Used by lower-level Widgets to locate and access the _ModelBindingState, which is the State Object for the ModelBinding StatefulWidget.

- CarModel
    - This represents the state for the app.

- Currently it holds the list of Car objects.
- The data in this class is immutable, it cannot be changed.
- To change the state in the app (the Car list), the ‘update’ method in the ModelBinding class must be invoked, passing in a new CarModel.

Conclusion

At first, I thought that the InheritedWidget would make life easy. I thought that you could “Reach Up the Tree and Get Data”: get data, update it and the UI would re-render itself. It doesn’t.

You can use InheritedWidget in a simple manner to hold non-mutating state data and access it from lower-level widgets.

However, if you want to hold mutating state data, update it and have the UI re-render itself, you have to wrap the InheritedWidget within a StatefulWidget and force the StatefulWidget to re-render the State Tree by calling the ‘setState’ method. A lot more complicated.

Further Reading

I highly recommend the following articles:
https://www.didierboelens.com/2018/06/widget---state-

--context---inheritedwidget/
https://stackoverflow.com/questions/49491860/flutter-how-to-correctly-use-an-inherited-widget
https://medium.com/flutter-io/managing-flutter-application-state-with-inheritedwidgets-1140452befe1

31. State & ScopedModel Approach

Introduction

I don’t think that the InheritedWidget approach turned out to be a good solution for our state issues. Once you added state / mutation and re-rendering of new state into account, it turned out a lot more complicated than expected.

The purpose of this chapter is to take a look at the ScopedModel approach.

Approach

This approach removes most of the requirements to use Stateful Widgets, enabling the user to use Stateless Widgets instead in many cases.

ScopedModel has been mentioned in many articles as an alternative to just using InheritedWidget. At first sight, it looks like the ScopedModel package is

basically InheritedWidget, only made easier to use.

Package

ScopedModel is a Dart package and it is available here: https://pub.dartlang.org/packages/scoped_model

As it is a package you will have to install it: https://pub.dartlang.org/packages/scoped_model - -installing-tab-

Package Readme

The package README.md file includes the following text:
A set of utilities that allow you to easily pass a data Model from a parent Widget down to its descendants. In addition, it also rebuilds all of the children that use the model when the model is updated. This library was originally extracted from the Fuchsia codebase.
This package provides three main classes:

1. Model
    - You will extend this class to create your own Models, such as SearchModel or UserModel.
    - You can listen to Models for changes!
2. ScopedModel Widget.
    - If you need to pass a Model deep down

your Widget hierarchy, you can wrap your Model in a ScopedModel Widget.
    - This will make the Model available to all descendant Widgets.
3. ScopedModelDescendant Widget.
    - Use this Widget to find the appropriate ScopedModel in the Widget tree.
    - It will automatically rebuild whenever the Model notifies that change has taken place.

Multiple Models

At first glance, it looks as if this package allows the user to use multiple State Models. This certainly makes it a better candidate for working with larger applications. You could have User data in one model, Transaction data in another etc.

Exercise – ‘state_and_scoped_model’

The code below is not perfect by any means (you can add the same car twice and when you tap on it, it selects both) but it demonstrates how to get an app up and working with ScopedModel and how you can maintain separate states in separate models.

In this exercise, I use the ScopedModel to handle two

separate state models:

1. a list of cars (to which we can add cars)
2. the currently selected car (which you can change by tapping on a car).

There is more code for you to copy and paste in this example. However, this app does more than some of the previous examples: it allows you to add cars and allows you to select cars.

Step 1 – Create Default Flutter App

Follow the instructions in Generate Your First App
Leave project open.

Step 2 – Replace Application Code

Replace contents of file ‘main.dart’ in folder ‘lib’ with the following:

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

void main() => runApp(new CarAppWidget());

class Car {
  String _make;
  String _model;
  String _imageSrc;
```

```
  Car(this._make, this._model, this._imageSrc);

  operator ==(other) =>
      (other is Car) && (_make == other._make) && (_model
== other._model);

  int get hashCode => _make.hashCode ^ _model.hashCode ^
_imageSrc.hashCode;
}

class CarListModel extends Model {
  List<Car> _carList = [
    Car(
      "Bmw",
      "M3",
      "Https://media.ed.edmunds-
media.com/bmw/m3/2018/oem/2018_bmw_m3_sedan_base_fq_o
    ),
    Car(
      "Nissan",
      "GTR",
      "Https://media.ed.edmunds-media.com/nissan/gt-
r/2018/oem/2018_nissan_gt-
r_coupe_nismo_fq_oem_1_150.jpg",
    ),
    Car(
      "Nissan",
      "Sentra",
```

```
      "Https://media.ed.edmunds-
media.com/nissan/sentra/2017/oem/2017_nissan_sentra_sedan_s
turbo_fq_oem_4_150.jpg",
    )
  ];

  List<Car> get carList => _carList;

  void add(String make, String model, String imageSrc) {
    _carList.add(Car(make, model, imageSrc));
    notifyListeners();
  }
}

class CarSelectionModel extends Model {
  Car _selectedCar;

  Car get selectedCar => _selectedCar;

  void set selectedCar(Car selectedCar) {
    _selectedCar = selectedCar;
    notifyListeners();
  }

  bool isSelected(Car car) {
    if (_selectedCar == null) {
      return false;
    } else {
      return car == _selectedCar;
```

```
    }
  }
}

class CarAppWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
        title: 'Car App',
        theme: new ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: ScopedModel<CarListModel>(
            model: CarListModel(),
            child: ScopedModel<CarSelectionModel>(
                model: CarSelectionModel(),
                child: CarAppLayoutWidget(title: 'Cars'))));
  }
}

class CarAppLayoutWidget extends StatelessWidget {
  CarAppLayoutWidget({Key key, this.title}) : super(key:
key);
  final String title;

  _addCar(BuildContext context) {
    ScopedModel.of<CarListModel>(context,
rebuildOnChange: true).add(
```

```
        "Subaru",
        "WRX",
        "Https://media.ed.edmunds-media"
        ".com/subaru/wrx/2018/oem/2018_subaru_wrx_sedan_sti-limited_s_oem_1_150"
        ".jpg");
  }

  String _calculateSelectedCarName(BuildContext context) {
    Car selectedCar =
        ScopedModel.of<CarSelectionModel>(context, rebuildOnChange: true)
            .selectedCar;

    if (selectedCar == null) {
      return "No car selected.";
    } else {
      return "Selected: ${selectedCar._make} ${selectedCar._model}";
    }
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
```

```
        body: Center(child: CarListWidget()),
        persistentFooterButtons: <Widget>[
          Text(_calculateSelectedCarName(context)),
          IconButton(
              icon: Icon(Icons.add),
              onPressed: () {
                _addCar(context);
              }),
        ]);
  }
}

class CarListWidget extends StatelessWidget {
@override
  Widget build(BuildContext context) {
    final carList =
        ScopedModel.of<CarListModel>(context,
rebuildOnChange: true).carList;
    List<CarWidget> carWidgets = carList.map((Car car) {
      return CarWidget(car);
    }).toList();
    return new ListView(children: carWidgets);
  }
}

class CarWidget extends StatelessWidget {
  CarWidget(this._car) : super();
```

```
  final Car _car;

  _buildCarWidget(context, child, CarSelectionModel
selectionModel) {
    return GestureDetector(
        onTap: () => selectionModel.selectedCar = _car,
        child: Padding(
            padding: EdgeInsets.all(20.0),
            child: Container(
                decoration: BoxDecoration(
                    border: Border.all(),
                    color: selectionModel.isSelected(_car)
                        ? Colors.blue
                        : Colors.white),
                padding: EdgeInsets.all(20.0),
                child: Center(
                    child: Column(children: <Widget>[
                  Text('${_car._make} ${_car._model}',
                      style: TextStyle(fontSize: 24.0)),
                  Padding(
                      padding: EdgeInsets.only(top: 20.0),
                      child: Image.network(_car._imageSrc))
                ])))));
  }

  @override
  Widget build(BuildContext context) {
```

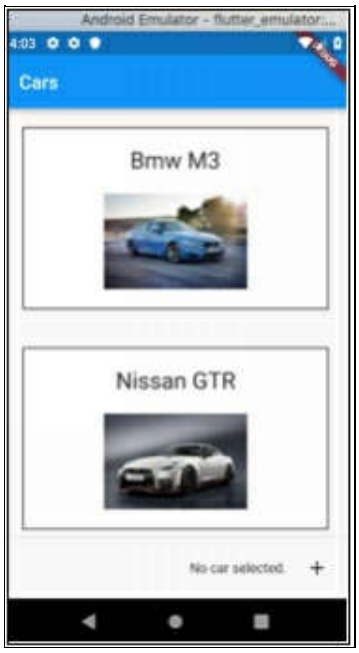
```
    return ScopedModelDescendant<CarSelectionModel>(
        builder: (context, child, selectionModel) =>
            _buildCarWidget(context, child, selectionModel));
  }
}
```

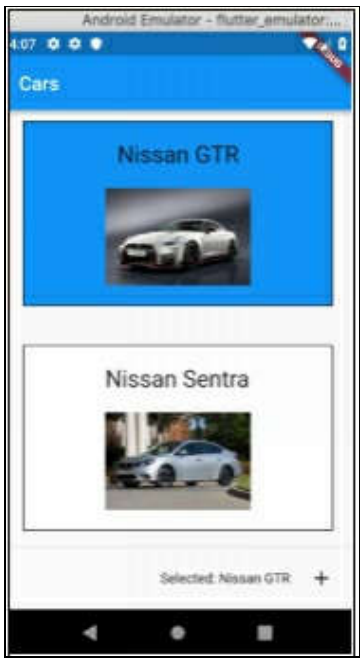
Step 3 – Open Emulator & Run

Follow the instructions in [Open Android Emulator & Run Your First App]

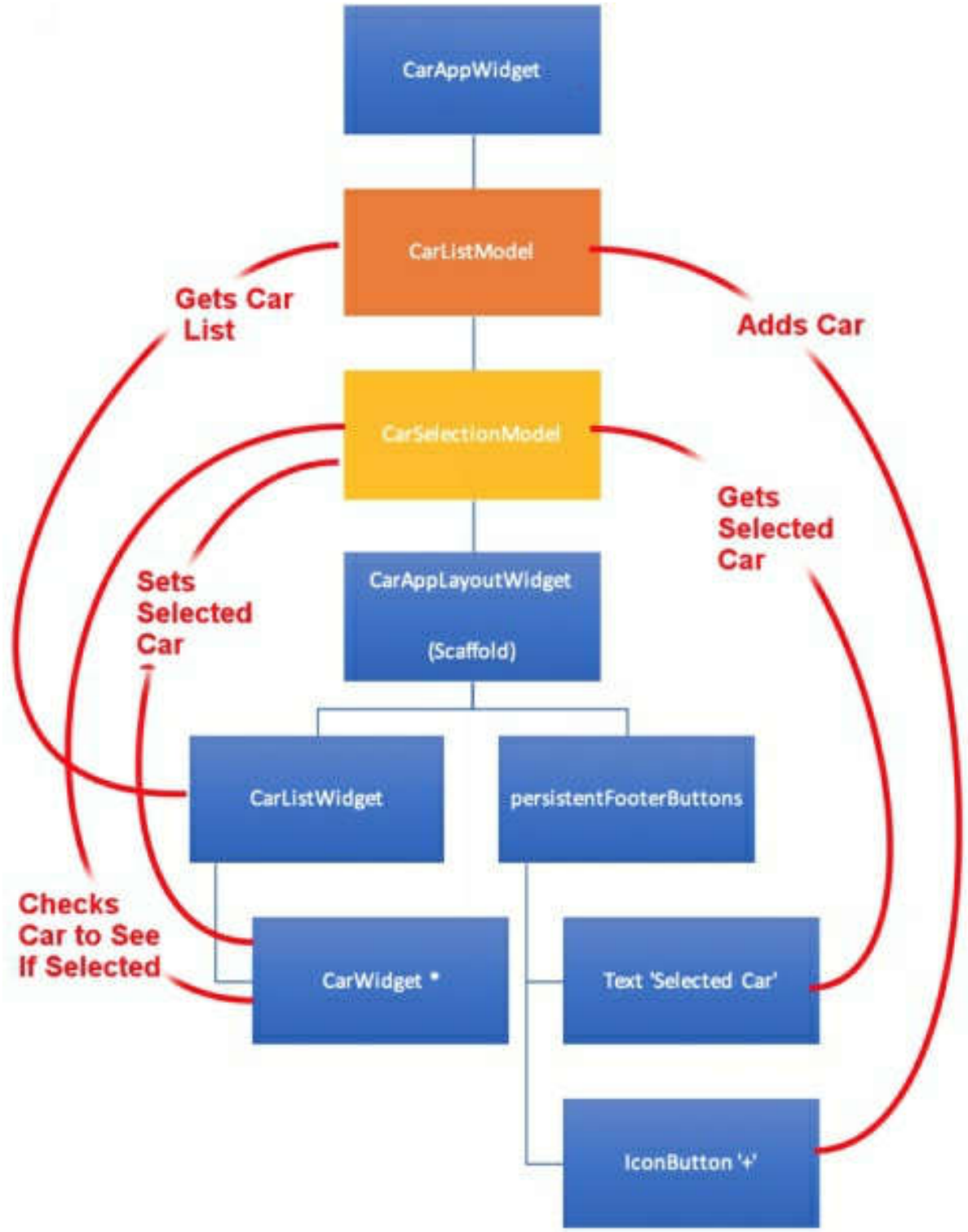
If you tap on the '+' button at the bottom it adds another car.

If you tap on a car it selects the car (adding a blue background) and sets the text of the selected car at the bottom.





Summary


CarAppWidget
CarListModel
Gets Car List
Adds Car
CarSelectionModel
Gets Selected Car
Sets Selected Car
CarAppLayoutWidget
(Scaffold)
CarListWidget
persistentFooterButtons
Checks Car to See If Selected
CarWidget *
Text 'Selected Car'
IconButton '+'

- CarListModel holds state for car list.
    - Note that the ‘add’ method add a car and that it calls ‘notifyListeners’ to ensure the children in the Widget Tree are updated.
- CarSelectionModel holds state for selected car.
    - Note that the ‘set’ method for the ‘selectedCar’ calls ‘notifyListeners’ to ensure the children in the Widget Tree are updated.
- CarListWidget is used to render car list. It gets its state from the CarListModel.
- CarWidget uses a ScopedModelDescendant from this package to use a builder to build the car widget. It gets the data for the car from the constructor. The ScopedModelDescendant enables the builder to get the selection state from the CarSelectionModel.
- CarAppLayoutWidget lays out the Widgets in a Scaffold.
    - PersistentFooterButtons is used to show Text and a Button at the bottom, even if the user scrolls.
        - The Text for the selected car name is calculated by calling ‘ScopedModel.of’ to get to the

CarSelectionModel and calling a method there to get the text.

- The ‘+’ Button calls ‘ScopedModel.of’ to get to the CarListModel and calls a method there to add a car to the list of cars.

Conclusion

I was impressed by this package; how simple it was to get going and how well it worked with multiple models. I really think this is the way to go for small / medium sized projects. It was easy to get to the models using builders or using the ‘ScopedModel.of’ method. Nice and flexible.

32. State & BLoCs w/Streams Approach

Introduction

BLoC stands for ‘Business Logic Components’.
It’s a pattern for state management recommended by Google developers.

The purpose of this chapter is to learn this pattern for state management.

BLoC Pattern

This pattern is about storing the app main state in a central place (a business logic object stored in a Stateful Widget) and having it communicate with the rest of the app’s Widgets using streams and RxDart. Note that this pattern uses InheritedWidget to store the Business Logic Component within a widget in the hierarchy.

Reactive Programming

Reactive Programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. It is all about asynchronously emitting data to these streams or

listening to those streams and doing something with the data (perform operations on it). To oversimplify things, Observable objects write to these streams and Subscribers listen to these streams. Operators do something with the stream data, like create it, transform it, filter it, combine it etc. It sounds complicated but it can make your code much simpler when you get the hang of it.

One great thing about streams is that you can use them to commutate between software components. For example, rather than have ‘Component 1’ directly call a method in ‘Component 2’ when something happens, you could have Component 2 subscribe to an event stream in Component 1. When something happens in Component 1, it posts to the event stream and Component 2 is notified and does something.

RxDart

The BLoC pattern uses the RxDart package.
RxDart is a reactive functional programming library for Google Dart, based on ReactiveX. Google Dart comes with a very decent Streams API out-of-the-box; rather than attempting to provide an alternative to this API, RxDart adds functionality on top of it. So basically, RxDart enhances the Dart support for Streams!

StreamBuilder

This approach uses the StreamBuilder class to build stateless child Widgets. StreamBuilder is a Widget that builds itself based on the latest update from a Stream. StreamBuilders listen for changes in streams and build Widgets when the stream data changes. Thus, your Widgets can update when the state changes and the state change is pushed to a stream.

Exercise – ‘state_and_block_with_streams’

In this exercise, we use a BLoC with states and streams to enable the user to re-order a list of customers.

Step 1 – Create Default Flutter App

Follow the instructions in Generate Your First App
Leave project open.

Step 2 – Add the RxDart Dependency

Add the following dependencies to your ‘pubspec.yaml’ file. After that you will need to do a ‘flutter packages get’ on the command line in the root of your project to download the dependencies.

```
dependencies:
```

```
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your
application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2
  rxdart: 0.18.1

dev_dependencies:
  flutter_test:
    sdk: flutter
```

Step 3 – Replace Application Code

Replace contents of file ‘main.dart’ in folder ‘lib’ with the following:

```
import 'dart:async';

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

class Customer {
 String _firstName;
 String _lastName;
 bool _upButton;
 bool _downButton;

  Customer(this._firstName, this._lastName) {
   _upButton = false;
   _downButton = false;
  }

  String get name => _firstName + " " + _lastName;
```

```
  bool get upButton => _upButton;

  set upButton(bool value) {
   _upButton = value;
  }

  bool get downButton => _downButton;

  set downButton(bool value) {
   _downButton = value;
  }

  operator ==(other) =>
    (other is Customer) &&
    (_firstName == other._firstName) &&
    (_lastName == other._lastName);

  int get hashCode => _firstName.hashCode ^
_lastName.hashCode;
}

class Bloc {
 // BLoC stands for Business Logic Component.
 List<Customer> _customerList = [];

  Bloc() {
   _upActionStreamController.stream.listen(_handleUp);
   _downActionStreamController.stream.listen(_handleDown);
  }

  List<Customer> initCustomerList() {
   _customerList = [
    new Customer("Fred", "Smith"),
    new Customer("Brian", "Johnson"),
    new Customer("James", "McGirt"),
    new Customer("John", "Brown")
   ];
   updateUpDownButtons();
   return _customerList;
  }
```

```
  void dispose() {
   _upActionStreamController.close();
   _downActionStreamController.close();
  }

  void _handleUp(Customer customer) {
   swap(customer, true);
   updateUpDownButtons();

    _customerListSubject.add(_customerList);
   _messageSubject.add(customer.name + " moved up");
  }

  void _handleDown(Customer customer) {
   swap(customer, false);
   updateUpDownButtons();

    _customerListSubject.add(_customerList);
   _messageSubject.add(customer.name + " moved down");
  }

  void swap(Customer customer, bool up) {
   int idx = _customerList.indexOf(customer);
   _customerList.remove(customer);
   _customerList.insert(up ? idx - 1 : idx + 1, customer);
  }

  void updateUpDownButtons() {
   //TODO We dont really need to update them all, but this is
just an example.
   for (int idx = 0, lastIdx = _customerList.length - 1;
      idx <= lastIdx;
      idx++) {
    Customer customer = _customerList[idx];
    customer.upButton = (idx > 0);
    customer.downButton = (idx < lastIdx);
   }
  }

  // Streams for State Updates
```

```
 Stream<List<Customer>> get customerListStream =>
_customerListSubject.stream;
 final _customerListSubject =
BehaviorSubject<List<Customer>>();

  Stream<String> get messageStream =>
_messageSubject.stream;
 final _messageSubject = BehaviorSubject<String>();

  // Sinks for Actions
 Sink<Customer> get upAction =>
_upActionStreamController.sink;
 final _upActionStreamController =
StreamController<Customer>();

  Sink<Customer> get downAction =>
_downActionStreamController.sink;
 final _downActionStreamController =
StreamController<Customer>();
}

class BlocProvider extends InheritedWidget {
 final Bloc bloc;

  BlocProvider({
   Key key,
   @required this.bloc,
   Widget child,
  }) : super(key: key, child: child);

  @override
 bool updateShouldNotify(InheritedWidget oldWidget) =>
true;

  static Bloc of(BuildContext context) =>
    (context.inheritFromWidgetOfExactType(BlocProvider) as
BlocProvider).bloc;
}

class CustomerWidget extends StatelessWidget {
```

```
 final Customer _customer;

 CustomerWidget(this._customer);

 @override
 Widget build(BuildContext context) {
  final bloc = BlocProvider.of(context);
  Text text = Text(_customer.name,
    style: const TextStyle(fontSize: 15.0, fontWeight:
FontWeight.bold));
  IconButton upButton = IconButton(
    icon: new Icon(Icons.arrow_drop_up, color: Colors.blue),
    onPressed: () {
     bloc.upAction.add(_customer);
    });
  IconButton downButton = IconButton(
    icon: new Icon(Icons.arrow_drop_down, color:
Colors.blue),
    onPressed: () {
     bloc.downAction.add(_customer);
    });
  List<Widget> children = [];
  children.add(Expanded(
    child: Padding(padding: EdgeInsets.only(left: 20.0), child:
text)));
  if (_customer.upButton) {
   children.add(upButton);
  }
  if (_customer.downButton) {
   children.add(downButton);
  }
  return Padding(
    padding: EdgeInsets.all(6.0),
    child: ClipRRect(
      borderRadius: BorderRadius.circular(8.0),
      child: Container(
        decoration: BoxDecoration(color: Colors.cyan[100]),
        child: Row(
```

```
            children: children,
            mainAxisAlignment: MainAxisAlignment.start))));
  }
}

void main() => runApp(new CustomerAppWidget());

class CustomerAppWidget extends StatelessWidget {
 // This widget is the root of your application.
 final Bloc _bloc = new Bloc();

  @override
 Widget build(BuildContext context) {
  return new MaterialApp(
    title: 'Flutter Demo',
    theme: new ThemeData(
     primarySwatch: Colors.blue,
    ),
    home: BlocProvider(
     bloc: _bloc,
     child: new CustomerListWidget(
      title: 'Flutter '
         'Demo Home Page',
      messageStream: _bloc.messageStream,
     ),
    ),
  );
 }
}

class CustomerListWidget extends StatelessWidget {
 CustomerListWidget({Key key, this.title, Stream<String>
this.messageStream})
    : super(key: key) {
  this.messageStream.listen((message) {
    _scaffoldKey.currentState.showSnackBar(SnackBar(
     content: Text(message),
     duration: Duration(seconds: 1),
    ));
```

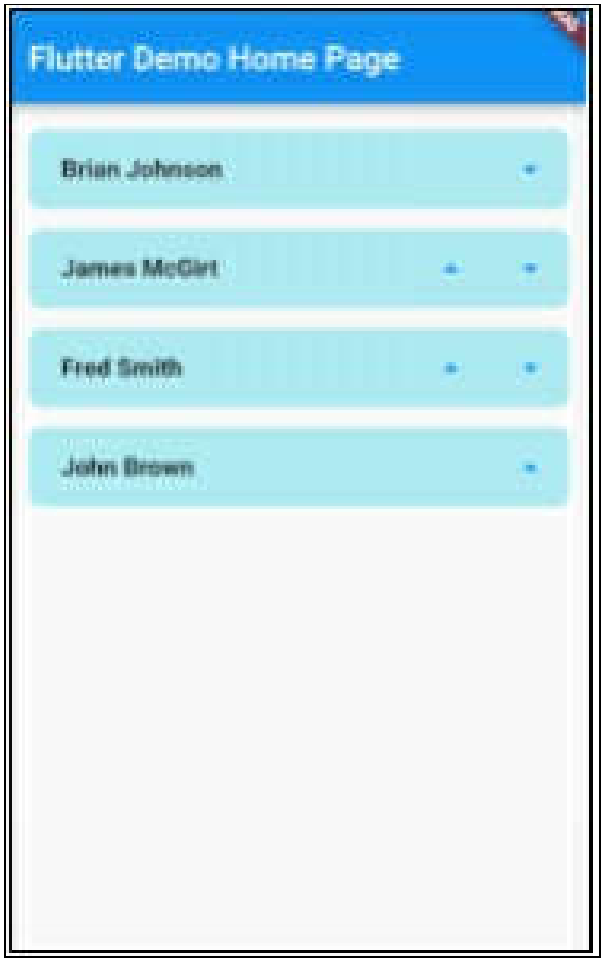
```
  });
 }

 final GlobalKey<ScaffoldState> _scaffoldKey =
GlobalKey<ScaffoldState>();
 final String title;
 final Stream<String> messageStream;

 @override
 Widget build(BuildContext context) {
  final bloc = BlocProvider.of(context);
  return new Scaffold(
    key: _scaffoldKey,
    appBar: new AppBar(
     title: new Text(title),
    ),
    body: StreamBuilder<List<Customer>>(
      stream: bloc.customerListStream,
      initialData: bloc.initCustomerList(),
      builder: (context, snapshot) {
       List<Widget> customerWidgets =
         snapshot.data.map((Customer customer) {
        return CustomerWidget(customer);
       }).toList();
       return ListView(
         padding: const EdgeInsets.all(10.0),
         children: customerWidgets);
      }));
 }
}
```

Step 4 – Open Emulator & Run

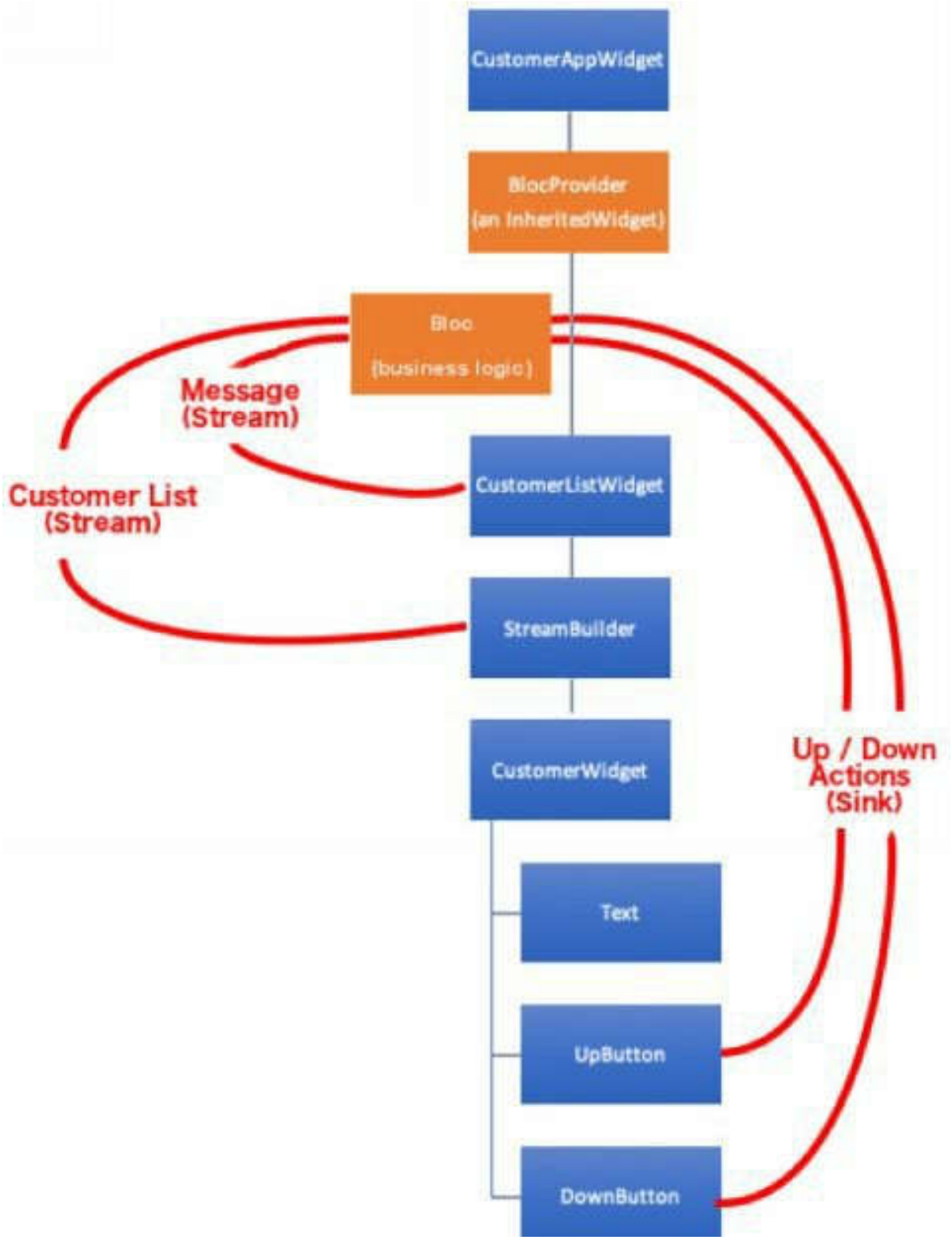
Follow the instructions in [Open Android Emulator & Run Your First App]

You can move the customers up and down using the

arrow icons.
Note that the user is also presented with a message at the bottom.



Summary



- CustomerAppWidget

    - Stateless Widget
    - Root of your application.
- BlocProvider
    - InheritedWidget
    - Wraps CustomerListWidget
    - Contains instance of Bloc object.
        - Has ‘of’ method to return instance of ‘Bloc’ to Widgets at lower levels of Widget tree.
- Bloc
    - Plain Dart class.
    - Business Logic Component.
    - Contains state (list of customer objects).
    - Contains 2 behavior subjects with streams.
        - Subject are something that can be observed. A BehaviorSubject is a subject that always provides the last emitted value from the stream, even if the subscription is added after that value was omitted.
        - Streams may be used to get an observable for a subject.
        - BehaviorSubjects and Streams are used to provide observable state to Widgets in the tree below.

    - Contains 2 stream controllers with sinks.
        - StreamControllers give you streams and a way to add events to the stream at any point, and from anywhere.
        - Sinks are generic destinations for data that can have values written to.
        - StreamControllers and Sinks are used here to listen for incoming data from a Widget event (customer clicks on up or down button).
- CustomerListWidget
    - Stateless Widget.
    - Contains list of customer widgets.
    - Has ‘message stream’ argument in constructor.
        - This is to listen to message stream in BLoC, displaying a message to the user every time the stream changes.
    - Has child StreamBuilder which listens to customer list stream in BLoC, returning a ListView of CustomerWidget objects every time the stream changes.

- CustomerWidget
    - Stateless Widget that draws a Customer with the name and up / down buttons.

Conclusion

This is a pattern rather than a package - you will have to implement the code yourself.
This looks straightforward.

- You could use multiple BLoCs in a single app to simplify a larger app. For example, you could have a CustomerBLoC, an OrderBLoC etc., just an InheritedWidget for each BLoC.
- You use a InheritedWidget to get access to the BLoC (or BLoCs) from anywhere in the Widget tree.

- You put the dynamic UI inside StreamBuilders, which listen to streams in the BLoC.
- Your event handling will write values to the Sinks to update the state.

I have used this pattern before and I think it works well. The only downside I see is you're your build methods have to use StreamBuilders when rendering