we start this tutorial by creating a new Flutter project and cleaning up all the comments from pubspec.yaml and main.dart.

We let our main.dart file be for this moment and start by adding the required dependencies to our pubspec.yaml.

**pubspec.yaml**

name: master\_detail

description: Responsive Master/Detail

publish\_to: 'none'

version: 1.0.0+1

environment:

sdk: ">=2.7.0 <3.0.0"

dependencies:

flutter:

sdk: flutter

equatable: ^1.1.1

flutter\_bloc: ^4.0.0

flutter:

uses-material-design: **true**

Equatable will help us to compare objects by its values rather than by reference, and flutter\_bloc will provide us with the framework we use for state management.

**A simple data class**

First, we need a class containing the data we want to show in the app. We keep this to an absolute minimum and create a class Item under a new directory data that has two attributes: name and detail.

**data/item.dart**

**import** 'package:equatable/equatable.dart';

**class** **Item** **extends** **Equatable** {

**final** String name;

**final** String detail;

Item(**this**.name, **this**.detail);

**factory** Item.fromItem(Item item){

**if** (item == **null**){

**return** **null**;

} **else** {

**return** Item(item.name, item.detail);

}

}

**@override**

List<Object> **get** props => [name, detail];

}

**Lines 9 – 15** define a copy constructor that does nothing else than creating a new instance of Item with the same attribute values as the instance passed via the parameter.

**Line 18** overrides the props getter that is required since Item extends Equatable. All elements of the list returned as props are used when comparing instances of Item.

**The Logic using the Bloc Pattern**

We are using the bloc (business logic component) pattern in this example.

First, create a new directory and call it bloc. In Android Studio right click on the directory → New → Bloc Generator → New Bloc and enter “master\_detail” as the name. Check “Do you want to use equatable” and click on ok. This will generate 4 files: bloc.dart, master\_detail\_event.dart, master\_detail\_state.dart, and master\_detail\_bloc.dart.

**bloc/bloc.dart**

**export** 'master\_detail\_bloc.dart';

**export** 'master\_detail\_event.dart';

**export** 'master\_detail\_state.dart';

This file simply allows to import these three files using one import statement: If you import bloc.dart, these files listed here are imported.

**bloc/master\_detail\_event.dart**

**import** 'package:equatable/equatable.dart';

**import** 'package:master\_detail/data/item.dart';

**abstract** **class** **MasterDetailEvent** **extends** **Equatable** {

**const** MasterDetailEvent();

}

**class** **LoadItemsEvent** **extends** **MasterDetailEvent** {

**@override**

List<Object> **get** props => [];

}

**class** **AddItemEvent** **extends** **MasterDetailEvent** {

**final** Item element;

AddItemEvent(**this**.element);

**@override**

List<Object> **get** props => [element];

}

**class** **SelectItemEvent** **extends** **MasterDetailEvent** {

**final** Item selected;

SelectItemEvent(**this**.selected);

**@override**

List<Object> **get** props => [selected];

}

In this file we create three Events that can be sent to our bloc to request a state change. All three Events extend our abstract class MasterDetailEvent and we again use Equatable for comparison.

The LoadItemsEvent notifies the bloc to load all existing items. AddItemEvent requests to add a new item specified by the event’s attribute element. SelectItemEvent will be used to notify the bloc that an Item has been selected in the list.

**bloc/master\_detail\_state.dart**

**import** 'package:equatable/equatable.dart';

**import** 'package:master\_detail/data/item.dart';

**abstract** **class** **MasterDetailState** **extends** **Equatable** {

**const** MasterDetailState();

}

**class** **LoadingItemsState** **extends** **MasterDetailState** {

**@override**

List<Object> **get** props => [];

}

**class** **NoItemsState** **extends** **MasterDetailState** {

**@override**

List<Object> **get** props => [];

}

**class** **LoadedItemsState** **extends** **MasterDetailState** {

**final** List<Item> elements;

**final** Item selectedElement;

LoadedItemsState(**this**.elements, **this**.selectedElement);

**@override**

List<Object> **get** props => [selectedElement, ...elements];

}

Where an event is the input to our bloc, a state is what our bloc will give us in return. For this example our app will have 3 different states: LoadingItemsState, NoItemsState and LoadedItemsState.

Depending on the state we will display different UI elements in our app. Where LoadingItemsState and NoItemsState do not contain any additional information, the LoadedItemsState holds a list of Items and the currently selected Item.

**bloc/master\_detail\_bloc.dart**

**import** 'dart:async';

**import** 'package:flutter\_bloc/flutter\_bloc.dart';

**import** 'package:master\_detail/data/item.dart';

**import** 'bloc.dart';

**class** **MasterDetailBloc** **extends** **Bloc**<**MasterDetailEvent**, **MasterDetailState**> {

List<Item> \_items = [];

Item \_selected;

**@override**

MasterDetailState **get** initialState => LoadingItemsState();

**@override**

Stream<MasterDetailState> mapEventToState(

MasterDetailEvent event,

) **async**\* {

**if** (event **is** AddItemEvent) {

\_items.add(event.element);

} **else** **if** (event **is** SelectItemEvent) {

\_selected = event.selected;

}

**yield**\* \_loadItems();

}

Stream<MasterDetailState> \_loadItems() **async**\* {

**if** (\_items.isEmpty) {

**yield** NoItemsState();

} **else** {

**final** newState = LoadedItemsState([...\_items], Item.fromItem(\_selected));

**yield** newState;

}

}

}

This class is the heart of our business logic.

Let’s have a look at the mapEventToState function. Here we receive all events that are sent to our block. The aim of the function is to figure out which state the app should have depending of the received events.

First, we check if the event is an AddItemEvent. In this case we add the received item (passed through the event as a parameter) to our list. Second, we check if the event is a SelectItemEvent. If this is the case we assign the selected element to our variable.

After handling any of the events we call the \_loadItems function. All it does is to create a new state depending if there are no items (NoItemsState) or there are some items stored in the bloc (LoadedItemsState).

Note that we do not explicitly handle the LoadItemsEvent. Since \_loadItems is called regardless of which event we receive, we do not need any other functionality to be executed when a LoadItemsEvent is received.

**The responsive Master-Detail UI**

Finally we are going to build the UI for our application. For the UI files create a new directory called ui. First we have a look at the files master.dart and detail.dart.

**ui/master.dart**

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

**import** 'package:flutter\_bloc/flutter\_bloc.dart';

**import** 'package:master\_detail/bloc/bloc.dart';

**import** 'package:master\_detail/data/item.dart';

**import** 'package:master\_detail/ui/detail.dart';

**class** **Master** **extends** **StatefulWidget** {

**@override**

\_MasterState createState() => \_MasterState();

}

**class** **\_MasterState** **extends** **State**<**Master**> {

int elementCount = 0;

MasterDetailBloc \_bloc;

**@override**

**void** initState() {

**super**.initState();

\_bloc = BlocProvider.of(context);

\_bloc.add(LoadItemsEvent());

}

**@override**

Widget build(BuildContext context) {

**return** Scaffold(

appBar: AppBar(

title: Text("Master"),

actions: <Widget>[

IconButton(icon: Icon(Icons.add), onPressed: \_addItem)

],

),

backgroundColor: Color(0xffefefef),

body: BlocBuilder(

bloc: \_bloc,

builder: (context, state) {

**if** (state **is** LoadingItemsState) {

**return** Center(child: CircularProgressIndicator());

} **else** **if** (state **is** NoItemsState) {

**return** Center(child: Text("No Items"));

} **else** **if** (state **is** LoadedItemsState) {

**return** ListView.builder(

itemCount: state.elements.length,

itemBuilder: (context, index) {

**final** item = state.elements[index];

**return** ListTile(

title: Text(item.name),

selected: item == state.selectedElement,

onTap: () => \_selectItem(context, item),

);

},

);

}

**throw** Exception("unexpected state $state");

},

),

);

}

\_addItem() {

**final** newItem = Item(

"name $elementCount",

"This is the detail for element $elementCount",

);

\_bloc.add(AddItemEvent(newItem));

elementCount++;

}

\_selectItem(BuildContext context, Item item) {

\_bloc.add(SelectItemEvent(item));

**if** (MediaQuery.of(context).size.shortestSide < 768) {

**final** route = MaterialPageRoute(builder: (context) => Detail());

Navigator.push(context, route);

}

}

}

We have a stateful widget Master here. In its state we retrieve the MasterDetailBloc we have created earlier (line 19) and use it in a BlocBuilder to create the UI depending on its state.

Here we check if the screen is smaller than 768 (which we will then classify as a smartphone). When selecting an item on a small screen size like this, we need to navigate to a different view to show the detailed information of the item. For a tablet this is not necessary, since we have plenty of space to show this information right next to the Master list.

**ui/detail.dart**

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

**import** 'package:flutter\_bloc/flutter\_bloc.dart';

**import** 'package:master\_detail/bloc/bloc.dart';

**class** **Detail** **extends** **StatefulWidget** {

**@override**

\_DetailState createState() => \_DetailState();

}

**class** **\_DetailState** **extends** **State**<**Detail**> {

MasterDetailBloc \_bloc;

**@override**

**void** initState() {

**super**.initState();

\_bloc = BlocProvider.of<MasterDetailBloc>(context);

}

**@override**

Widget build(BuildContext context) {

**return** Scaffold(

appBar: AppBar(

title: Text("Detail"),

),

body: BlocBuilder(

bloc: \_bloc,

builder: (context, state) {

**if** (state **is** LoadedItemsState) {

**return** Center(

child: Text(state.selectedElement?.detail ?? "No item selected"),

);

} **else** {

**return** Container();

}

},

),

);

}

}

The Detail widget looks very similar to the Master widget. It also retrieves the MasterDetailBloc and uses BlocBuilder to build a state-dependent UI. Here we use the LoadedItemsState‘s selectedElement attribute to get the detailed information to display.

**ui/home\_page.dart**

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

**import** 'package:master\_detail/ui/detail.dart';

**import** 'package:master\_detail/ui/master.dart';

**class** **HomePage** **extends** **StatelessWidget** {

**@override**

Widget build(BuildContext context) {

**return** LayoutBuilder(

builder: (context, constraints) {

**if** (constraints.maxWidth > 768) {

**return** \_TabletHomePage();

} **else** {

**return** \_MobileHomePage();

}

},

);

}

}

**class** **\_MobileHomePage** **extends** **StatelessWidget** {

**@override**

Widget build(BuildContext context) {

**return** Master();

}

}

**class** **\_TabletHomePage** **extends** **StatelessWidget** {

**@override**

Widget build(BuildContext context) {

**return** Row(

children: <Widget>[

Container(width: 300, child: Master()),

Expanded(child: Detail())

],

);

}

}

For our main screen (HomePage) we use a LayoutBuilder to decide if we want to display the tablet or the mobile version of our screen. As in the master.dart file we set our breakpoint to 768 pixel to decide whether or not the device is a tablet.

For the mobile version of the HomePage (\_MobileHomePage) we only display the Master widget. For the tablet version (\_TabletHomePage) we show both: Master and Detail. We limit the width of the Master widget to 300px and let the Detail take the rest of the space.

**main.dart**

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

**import** 'package:flutter\_bloc/flutter\_bloc.dart';

**import** 'package:master\_detail/bloc/bloc.dart';

**import** 'package:master\_detail/ui/home\_page.dart';

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

**class** **MyApp** **extends** **StatelessWidget** {

**@override**

Widget build(BuildContext context) {

**return** BlocProvider(

create: (context) => MasterDetailBloc(),

child: MaterialApp(home: HomePage()),

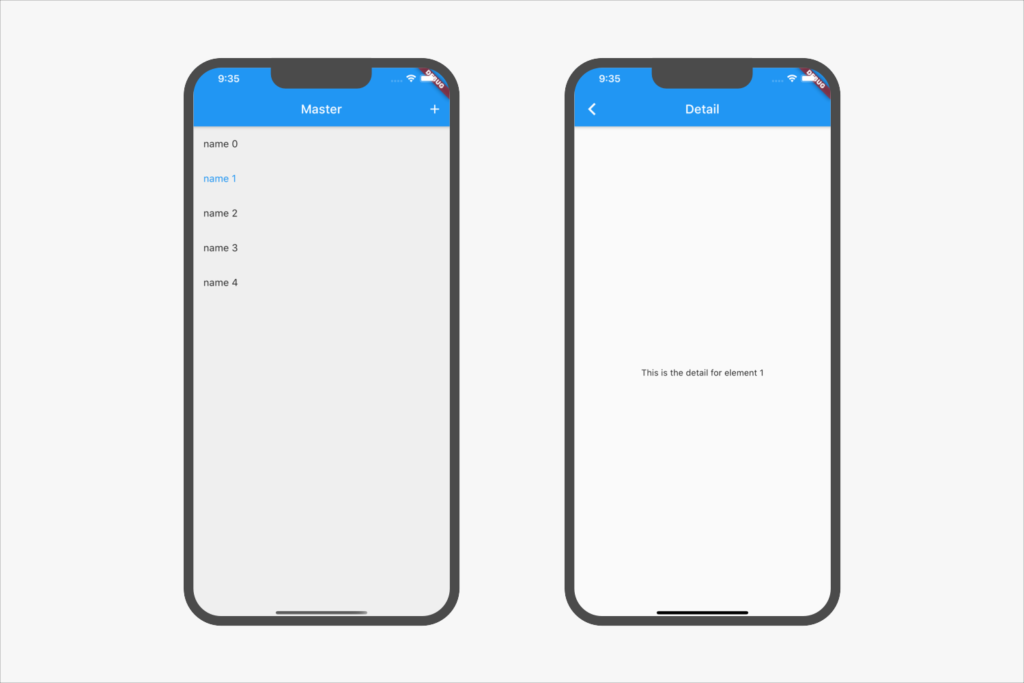
);

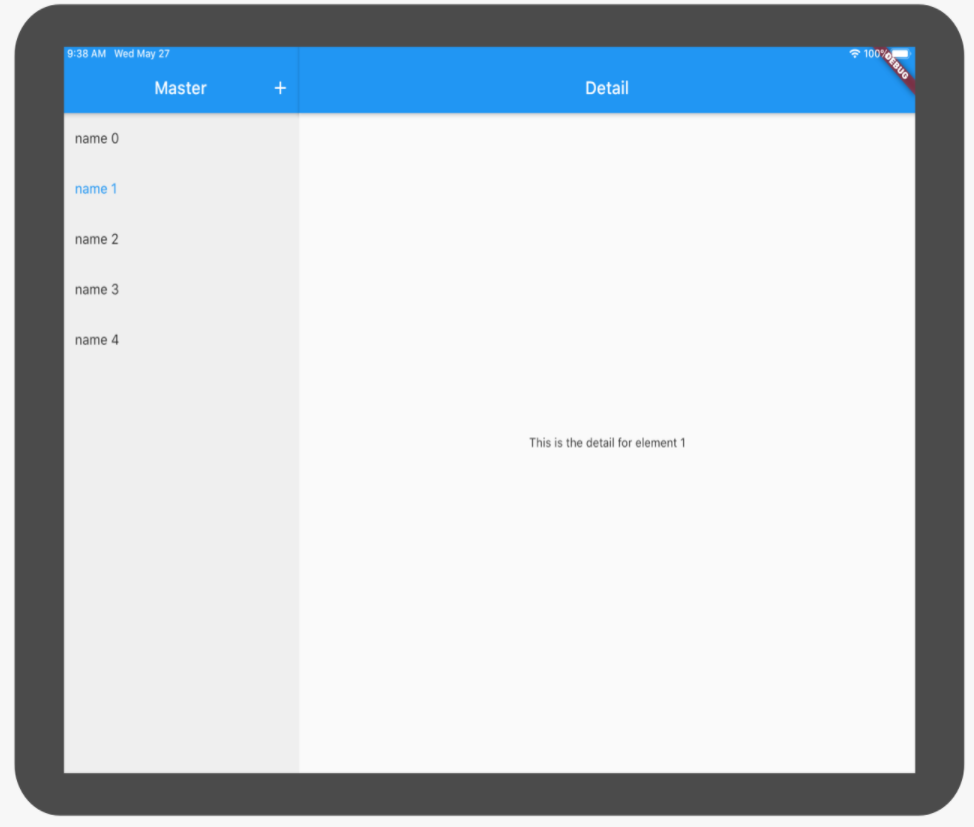
}

}

Finally, all that is left is to adapt the main.dart file to run our application. We use BlocProvider to make our MasterDetailBloc available throughout the widget tree and create a very basic MaterialApp showing our HomePage.

Now when we run the app on a smartphone the app uses two separate views to display the master (overview) widget and the detail widget. On a tablet we are making use of the additional screen space to display both widgets next to each other.

On smaller screens the application is displayed using two separate screens to show the master and the detail view.



**On tablets we have both views combined in one screen.**