Sommario

[Introduction 2](#_Toc88267235)

[1. State management solutions presentation 3](#_Toc88267236)

[1.1 SetState and InheritedWidget/InheritedModel 3](#_Toc88267237)

[1.2 Redux 3](#_Toc88267238)

[1.3 Bloc 3](#_Toc88267239)

[1.4 MobX 3](#_Toc88267240)

[1.5 GetX 3](#_Toc88267241)

[2. The Todo app 4](#_Toc88267242)

[2.1 General overview 4](#_Toc88267243)

[2.1.1 Base functionalities 4](#_Toc88267244)

[2.1.2 Adding new features 5](#_Toc88267245)

[2.1.3 Renders optimization 5](#_Toc88267246)

[2.2 Implementation 6](#_Toc88267247)

[2.2.1 Shared project structure and files 6](#_Toc88267248)

[2.2.2 Inherited widget/model and SetState implementation 13](#_Toc88267249)

[2.2.3 Redux 29](#_Toc88267250)

[2.2.4 BLoC 36](#_Toc88267251)

[2.2.5 MobX 57](#_Toc88267252)

[2.2.6 GetX 68](#_Toc88267253)

[3. The other app 68](#_Toc88267254)

[4. Comparisons 69](#_Toc88267255)

[5. Conclusions 69](#_Toc88267256)

# Introduction

A short introduction

# State management solutions presentation

I will present here the main concepts of every solution

## SetState and InheritedWidget/InheritedModel

Main concepts of setState and inheritedWidgets..

## Redux

Main concepts of Redux

## Bloc

Main concepts of bloc

## MobX

Main concepts of MobX

## GetX

Main concepts of GetX

# The Todo app

In this section I’m going to present the development of a Todo management mobile app using a series of State Management solutions and collect some measurements. In particular for every solution three development processes will be executed. First will be implemented the basic functionalities of the app. Then I will measure how much effort/code is needed to add other functionalities. In the last round some optimization will be made to the code in terms of UI renders and memory consumption.

## General overview

The development process is divided in three parts.

### Base functionalities

It offers the possibility to visualize and partially handle todos. It is composed of a single page: the HomePage. The HomePage is composed by an appbar and two tabs: the todo tab and the stats tab.

In the todo tab the list of todos is visualized. Is possible to filter the todo using a DropdownButton in the top right corner inside the AppBar.

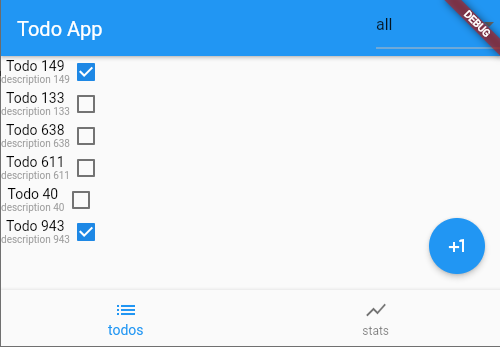
The possible filter values are:

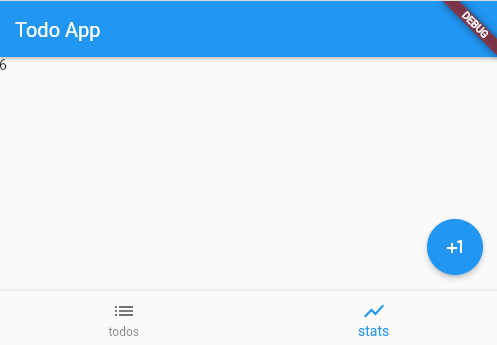
* **All** (visualize completed and pending todos)
* **Completed** (visualize completed todo)
* **Not Completed** (visualize pending todos)

The elements inside the list of todos are called TodoItems. TodoItems visualize the todo’s name and description using a Text widget and completion using a Checkbox. It is possible to use the checkbox to mark a Todo as completed or to mark it as pending.

In the stats Tab instead is possible to visualize the number of completed todos through a Text widget.

In the lower part a TabSelector allow to switch from tabs.





### Adding new features

Once basic functionalities got implemented a few more will be added as said above. In particular the AddTodo feature and the UpdateTodo feature will be added.

#### The Add todo Feature

This is a simple feature. It adds the possibility to create new Todos using the floatingbutton in the bottom right corner.

Immagine che contiene testo

Descrizione generata automaticamente

#### The Update feature

This feature allow to tap on a TodoItem to navigate to another route/page where a TextField and a confirm button will be present. Once inserted the new name for the todo clicking on the confirm button the route will be popped and the todo will be updated. This is a slightly difficult feature with respect to the add one for the fact we are going to pass the state from one route7tree to another.

Immagine che contiene testo

Descrizione generata automaticamente

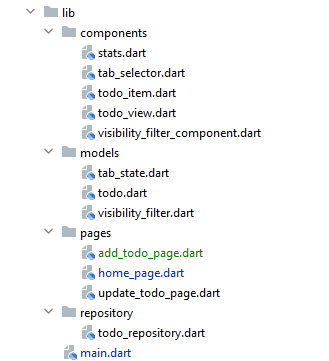
### Renders optimization

In this part some optimization for the widgets rendering will be made. In particular the aim is to use the least rerenders possible. The focus will be on the TodoView and TodoItems. We want the TodoView to rerender only when a structure change happends in the filteredTodo list and not on a single TodoItem’s internal aspects change. In other word when Todo are modified through the completion feature or the update feature only the corresponding TodoItem should be rerendered leaving the rest of the application intact.

## Implementation

### Shared project structure and files

Some parts of the code will be shared/reused between solution’s implementations. The complete structure of the app’s UI will be the same for every implementation such as the repository used to fetch Todos. On the following part the shared UI implementation will be presented.



The figure 2.2.3 illustrate the folder and files structure.

#### main.dart

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

class MyApp extends StatefulWidget {  
 const MyApp({Key? key}) : super(key: key);  
  
 @override  
 State<MyApp> createState() => \_MyAppState();  
}  
  
class \_MyAppState extends State<MyApp> {  
  
 @override  
 Widget build(BuildContext context) {  
 return MaterialApp(  
 initialRoute: "/",  
 routes: {  
 "/": (context) => const HomePage(),  
 "/updateTodo": (context) => UpdateTodoPage(),

"/addTodo": (context) => AddTodoPage(),  
  
 },  
 );  
 }  
}26

The entire app is wrapped into a MaterialApp widget with three routes. The HomePage , the UpdateTodoPage and the AddTodoPage.

#### Models and Repository

The **tab\_state.dart** file

enum TabState{  
 todos,stats  
}3

Tab states represent the possible tabs in the homepage.

**visibility\_filter.dart**

enum VisibilityFilter{

completed,notCompleted,all

}3

**todo.dart**

@immutable  
class Todo {  
 final int id;  
 final String name;  
 final String description;  
 final bool completed;  
  
 const Todo(  
 {required this.id,  
 required this.name,  
 required this.description,  
 required this.completed});  
  
 @override  
 bool operator ==(Object other) {  
 return (other is Todo) &&  
 other.description == description &&  
 other.name == name &&  
 other.id == id &&  
 other.completed == completed;  
 }  
  
 @override  
 String toString() {  
 return "{ id: $id completed: $completed}";  
 }  
  
 @override  
 *// TODO: implement hashCode* int get hashCode => super.hashCode;  
}31

Todo model can change in different implementations. It is presented here as an immutable class but in some implementation is will be change to mutable.

**todo\_repository.dart**

class TodoRepository {  
 static Future<List<Todo>> *loadTodos*() async {  
 Random rand = Random();  
 List<Todo> todos = [];  
 List<int> ids = [];  
 while (ids.length < 6) {  
 int newInt = rand.nextInt(1000)+2;  
 if (!ids.contains(newInt)) {  
 ids.add(newInt);  
 }  
 }  
 todos = ids  
 .map((number) => Todo(  
 id: number,  
 name: "Todo " + number.toString(),  
 description: "description " + number.toString(),  
 completed: rand.nextBool()))  
 .toList();  
  
 await Future.delayed(const Duration(seconds: 2));  
 return todos;  
 }  
  
 static Future<void> *saveTodos*(List<Todo> todos) async {  
 await Future.delayed(const Duration(seconds: 2));  
 }  
}27

TodoRepository has two static functions that simulate the loading and saving of todo to the Database. Those functions are async function with a duration of 2 seconds.

#### Pages

**home\_page.dart**

class HomePage extends StatefulWidget {  
 const HomePage({Key? key}) : super(key: key);  
  
 @override  
 State<HomePage> createState() => \_HomePageState();  
}  
  
class \_HomePageState extends State<HomePage> {  
 TabState tab = TabState.todos;   
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 actions: [  
 tab == TabState.todos  
 ? const VisibilityFilterComponent()  
 : Container()  
 ],  
 title: const Text("Todo App"),  
 ),  
 body: tab == TabState.todos ? const TodoView() : const Stats(),  
 bottomNavigationBar: TabSelector(  
 currTab: tab,  
 onTabChange:,  
 ),  
 floatingActionButton: tab == TabState.todos  
 ? FloatingActionButton(  
 child: const Icon(Icons.*plus\_one*),  
 onPressed: () {},  
 ) : null,  
 )  
 );  
 }  
}35

Homepage uses a simple Scaffold widget. The AppBar contains a VisibilityFilterComponent only when the tab is set to Todos. The body can change from todos to stats tab using the bottomNaviagationBar TabSelector. An empty FloatingActionButton is present for future implementation.

(note: some small pieces could change in different solution’s implementation. in the above example the tab changing is implemented through setState but it will not be always the case. Also the HomePage can be muted to Stateless widget in other implementations.).

**update\_todo\_page.dart**

class UpdateTodoPage extends StatefulWidget {  
 final Todo todo;  
  
 const UpdateTodoPage({Key? key, required this.todo}) : super(key: key);  
  
 @override  
 State<UpdateTodoPage> createState() => \_UpdateTodoPageState();  
}  
  
class \_UpdateTodoPageState extends State<UpdateTodoPage> {  
 final textController = TextEditingController();  
  
 @override  
 Widget build(BuildContext context) {  
  
 return Scaffold(  
 appBar: AppBar(  
 title: const Text("Update Todo"),  
 ),  
 body: Column(  
 children: [  
 TextField(  
 controller: textController,  
 decoration: const InputDecoration(  
 border: OutlineInputBorder(), hintText: 'Enter a new name'),  
 ),  
 TextButton(onPressed: (){}, child: const Text("Confirm"))  
 ],  
 ));  
 }  
  
 @override  
 void dispose() {  
 textController.dispose();  
 super.dispose();  
 }  
}37

The update todo page uses a Scaffold widget. The body is composed by a Column with inside a TextField and a TextButton. The TextButton is left empty for future implementation.

**add\_todo\_page.dart**

class AddTodoPage extends StatefulWidget {  
  
 const AddTodoPage({Key? key, required this.addTodoCallback}) : super(key: key);  
  
 @override  
 State<AddTodoPage> createState() => \_UpdateTodoPageState();  
}  
  
class \_UpdateTodoPageState extends State<AddTodoPage> {  
 final textControllerName = TextEditingController();  
 final textControllerDesc = TextEditingController();  
  
 @override  
 Widget build(BuildContext context) {  
  
 return Scaffold(  
 appBar: AppBar(  
 title: const Text("Add Todo"),  
 ),  
 body: Column(  
 children: [  
 TextField(  
 controller: textControllerName,  
 decoration: const InputDecoration(  
 border: OutlineInputBorder(), hintText: 'Enter a name'),  
 ),  
 TextField(  
 controller: textControllerDesc,  
 decoration: const InputDecoration(  
 border: OutlineInputBorder(), hintText: 'Enter a description'),  
 ),  
 TextButton(onPressed: (){},

child: const Text("Create"))  
 ],  
 ));  
 }  
  
 @override  
 void dispose() {  
 textControllerName.dispose();  
 textControllerDesc.dispose();  
 super.dispose();  
 }  
}46

The add\_todo\_page uses a Scaffold widget. The body is composed by a Column with inside two TextField widgets and a TextButton widget. The TextButton is left empty for future implementation.

#### Components

**todo\_view.dart**

class TodoView extends StatelessWidget {  
  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building TodoView");  
  
  
 return ListView.builder(  
 itemCount:,  
 itemBuilder: (context, index) {  
 return TodoItem(  
   
 );  
 },  
 );  
 }  
}19

TodoView uses a ListView. itemCount and itemBuilder fields are left empty for future implementation.

**todo\_item.dart**

class TodoItem extends StatelessWidget {  
 final Todo todo;  
  
 const TodoItem({Key? key, required this.id}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Todo Item $todo");  
  
 return InkWell(  
 onTap: () {  
 Navigator.*pushNamed*(context, "/updateTodo");  
 },  
 child: Row(  
 children: [  
 Column(  
 children: [  
 Text(todo.name,  
 style: const TextStyle(fontSize: 14, color: Colors.*black*)),  
 Text(todo.description,  
 style: const TextStyle(fontSize: 10, color: Colors.*grey*)),  
 ],  
 ),  
 Checkbox(  
 value: todo.completed,  
 onChanged: (value) {}),  
 ],  
 ),  
 );  
 }  
}31

TodoItem is a stateless widget. Uses two Text widgets to display the Todo information and a Checkbox to change the Todo’s completion. It is wrapped in a InkWell widget to make is responsive to taps. Functions are empty for future implementation.

**tab\_selector.dart**

class TabSelector extends StatelessWidget {  
  
  
 const TabSelector(  
 {Key? Key})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Tab Selector");  
  
 return BottomNavigationBar(  
 currentIndex: ,  
 onTap: (){},  
 items: TabState.values  
 .map((tab) => BottomNavigationBarItem(  
 label: describeEnum(tab),  
 icon: Icon(  
 tab == TabState.todos ? Icons.*list* : Icons.*show\_chart*,  
 ),  
 ))  
 .toList(),  
 );  
 }  
}25

Tabselector uses a BottomNavigationBar with as many BottomNavigationBarItems as TabState.values (in our case two). Function fields are left empty for future implementation.

**visibility\_filter\_component.dart**

class VisibilityFilterComponent extends StatelessWidget {  
  
 const VisibilityFilterComponent(  
 {Key? key})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Visibility filter");  
 return DropdownButton<VisibilityFilter>(  
 value:,  
 items: VisibilityFilter.values.map((filter) {  
 return DropdownMenuItem<VisibilityFilter>(  
 child: Text(describeEnum(filter)), value: filter);  
 }).toList(),  
 onChanged: (filter) {  
   
 },  
 );  
 }  
}21

VisibilityFilterComponent uses a DropdownButton with as many DropwodnMenuItems as VisibilityFilter.values (in our case two). Function fields are left empty for future implementation.

**stats.dart**

class Stats extends StatelessWidget {  
 const Stats({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Stats");  
  
 return Text();  
 }  
}10

Stats component is only a Text widget showing stats value.

**Lines of code written/updated: 314**

**Classes/widget created: 13**

### Inherited widget/model and SetState implementation

In this section Todo app will be implemented using two standard tools Flutter’s framework provides to handle state: **Inherited widget** (or the more advanced **InheritedModel**) and **setState**.

#### **State management solution’s introduction**

**setState** method notify the framework that the internal state of this object has changed.

Whenever you change the internal state of a [State](https://api.flutter.dev/flutter/widgets/State-class.html) object, make the change in a function that you pass to [setState](https://api.flutter.dev/flutter/widgets/State/setState.html).

setState(() { \_myState = newValue; });

The provided callback is immediately called synchronously. It must not return a future (the callback cannot be async), since then it would be unclear when the state was actually being set.

Calling [setState](https://api.flutter.dev/flutter/widgets/State/setState.html) notifies the framework that the internal state of this object has changed in a way that might impact the user interface in this subtree, which causes the framework to schedule a [build](https://api.flutter.dev/flutter/widgets/State/build.html) for this [State](https://api.flutter.dev/flutter/widgets/State-class.html) object.

If you just change the state directly without calling [setState](https://api.flutter.dev/flutter/widgets/State/setState.html), the framework might not schedule a [build](https://api.flutter.dev/flutter/widgets/State/build.html) and the user interface for this subtree might not be updated to reflect the new state.

**Inherited widget** are a base class for widgets that efficiently propagate information down the tree.

To obtain the nearest instance of a particular type of inherited widget from a build context, use *[BuildContext.dependOnInheritedWidgetOfExactType](https://api.flutter.dev/flutter/widgets/BuildContext/dependOnInheritedWidgetOfExactType.html).*

Inherited widgets, when referenced in this way, will cause the consumer to rebuild when the inherited widget itself changes state.

The convention is to provide a static method *of* on the [InheritedWidget](https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html) which does the call to [BuildContext.dependOnInheritedWidgetOfExactType](https://api.flutter.dev/flutter/widgets/BuildContext/dependOnInheritedWidgetOfExactType.html). This allows the class to define its own fallback logic in case there isn't a widget in scope. In the example above, the value returned will be null in that case, but it could also have defaulted to a value.

An [InheritedWidget](https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html) that's intended to be used as the base class for models whose dependents may only depend on one part or "aspect" of the overall model. An inherited widget's dependents are unconditionally rebuilt when the inherited widget changes.

**InheritedModel** widget is similar except that dependents aren't rebuilt unconditionally.

Widgets that depend on an [InheritedModel](https://api.flutter.dev/flutter/widgets/InheritedModel-class.html) qualify their dependence with a value that indicates what "*aspect*" of the model they depend on. When the model is rebuilt, dependents will also be rebuilt, but only if there was a change in the model that corresponds to the aspect they provided.

#### **Base app**

InheritedWidget requires to create a class where the state or part of the state will be contained and extends it to InheritedWidget. For our purpose a single class will be enough to contains all the state information. This class will be called TodoInheritedData.

class TodoInheritedData extends InheritedWidget{

Data that should be accessible down the tree must be placed inside it. In our case the only data needed is : a list of Todos, a VisibilityFilter , a Int for the stats ( for conciseness it will represent the number of completed todos) and another list of Todos that will contain the todos matching the filter. Inside the constructor final variables are initialized with the corresponding arguments and *stats* and *filteredTodos* list are computed. *filterTodo* function is just a function that takes the full list of todos and a filter and returns the filtered list. Important to notice is the fact that a *child* widget must be also passed in the constructor. This is because our TodoInheritedData is nothing else than a widget itself that wraps the data and make them accessible in the child tree.

class TodoInheritedData extends InheritedWidget{

final List<Todo> todos;  
 final List<Todo> filteredTodos;

final VisibilityFilter filter;

final int stats;

TodoInheritedData(  
 {   
 Key? key,  
 required this.todos,  
 required this.filter,  
 required Widget child})  
 : stats = todos.length,  
 filteredTodos = filterTodo(todos, filter),  
 super(child: child, key: key);

}

Is important to understand that TodoInheritedData widget is stateless. It cannot be changed (every value is final) but instead a new TodoInheritedData widget must be provided when a data change occurs.

The *updateShouldNotify* function must be overridden inside TodoInheritedData to avoid ui rebuilding when a new state without actual data changes occurs. Once a TodoInheritedData element is replaced with a new one this new element will take care to call the *updateShouldNotify* function to decide whether is necessary or not to notify changes in the subtree. If the function returns *true* the subtree is rebuilt, if return *false* instead is not.

@override  
bool updateShouldNotify(TodoInheritedData oldWidget) {  
 return !listEquals(oldWidget.filteredTodos, filteredTodos);  
}

In our case the *listEquals* function takes as parameters the old *filteredTodos* list and the new one and compare them element by element checking if changes were made. In the particular case no changes were performed it returns *true* and will lead the *updateShouldNotify* function to return *false* and not to rebuild the entire subtree.

At this point our TodoInheritedData can be used it in a stateful widget.

class TodoProvider extends StatefulWidget {  
 const TodoProvider({Key? key, required this.child}) : super(key: key);  
  
 final Widget child;  
  
 @override  
 \_TodoProviderState createState() => \_TodoProviderState();  
}

class \_TodoProviderState extends State<TodoProvider> {  
 List<Todo> todos = [];  
 VisibilityFilter filter = VisibilityFilter.all;

@override  
Widget build(BuildContext context) {  
 return TodoInheritedData(  
 todos: todos,  
 filter: filter,  
 child: widget.child,  
 );  
}

Note that the VisibilityFilter *filter* is set as *all* by default as convention.

We add also an *init* method to fetch the data from the repository on widget’s creation.

@override  
void initState() {  
 TodoRepository.*loadTodos*().then((todos) {  
 setState(() {  
 this.todos = todos;  
 });  
 });  
 super.initState();  
}

*loadTodos* is a TodoRepository’s async function that simulate the retrieval of the todos from a database.

We need to declare also the *of* method to retrieve our TodoInheritedData down the tree. This method is called is static and just extract the nearest TodoInheritedData element up in the tree using *dependOnInheritedWidgetOfExactType* method.

static TodoInheritedData? *of*(BuildContext context) {  
 final TodoInheritedData? result = context.dependOnInheritedWidgetOfExactType<TodoInheritedData>();

assert(result != null, 'No TodoInheritedData found in context');

return result;

}

At this point our TodoProvider widget can be incorporated as a parent of the Scaffold widget in the homepage. The usage of the Builder widget is due to the fact that data is accessible only in a context where a TodoProvider is already present. In other word TodoProvider’s data cannot be used in the same build method where it was instantiated into. Two options are possible; creating a separated file where to put our Scaffold or use a Builder widget that takes the current context and creates another with a TodoProvider widget.

@override  
Widget build(BuildContext context) {  
 return TodoProvider(  
 child: Builder(  
 builder: (context) {  
 return Scaffold(); }  
 );  
}

At this point the TodoView component can be populated. It is a stateless widget that will look up for the *filteredTodos* list in the TodoInheritedData inside the build method and create a ListView dynamically with it. The ListView will be composed by TodoItem widgets.

class TodoView extends StatelessWidget {  
  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building TodoView");  
  
 final List<Todo> filteredTodos = TodoInheritedData.*of*(context).filteredTodos;  
  
 return ListView.builder(  
 itemCount: filteredTodos.length,  
 itemBuilder: (context, index) {  
 return TodoItem(  
 todo: filteredTodos.elementAt(index),  
 );  
 },  
 );  
 }  
}

TodoItem widget is stateless widget that take as paramenter a Todo and take care of displaing it with the structure defined in at page x.

class TodoItem extends StatelessWidget {  
 final Todo todo;  
  
 const TodoItem({Key? key, required this.id}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return Row(  
 children: [  
 Column(  
 children: [  
 Text(todo.name,  
 style: const TextStyle(fontSize: 14, color: Colors.*black*)),  
 Text(todo.description,  
 style: const TextStyle(fontSize: 10, color: Colors.*grey*)),  
 ],  
 ),  
 Checkbox(  
 value: todo.completed,  
 onChanged: (value) {  
 }),  
 ],  
 ),  
 );  
 }  
}

At this point we got a single page (Homepage) that contains a TodoView showing *filteredTodos* list’s todos contained in the TodoInheritedData inside a TodoProvider widget. When the application starts we first see and empty page (todo are empty at the beginning) and then after few seconds a list of todos with their names, descriptions and completions appears. A list of fitered todos can be visualized but is not interactable yet.

In the app HomePage’s AppBar we already set up a VisibilityFilterComponent that is nothing else than a stateless widget. In its *build* method a DropdownButton’s value field is set up looking up for the filter values in the TodoInheritedData. Then the *items* field is filled with a list of DropdownMenuItem that comes from the mapping of all possible VisibilityFilter values to DropdownMenuItems.

class VisibilityFilterComponent extends StatelessWidget {  
  
 const VisibilityFilterComponent(  
 {Key? key})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Visibility filter");  
 VisibilityFilter filter= TodoInheritedData.*of*(context).filter;  
 return DropdownButton<VisibilityFilter>(  
 value: filter,  
 items: VisibilityFilter.values.map((filter) {  
 return DropdownMenuItem<VisibilityFilter>(  
 child: Text(describeEnum(filter)), value: filter);  
 }).toList(),  
 onChanged: (filter) {  
   
 },  
 );  
 }  
}

For what concerns the *onChanged* field a function that takes as single parameter a filter value must be provided. In particular we want this function to change the state contained in the TodoInheritedData (the filter part) and to fire a rebuild of the TodoInheritedData subtree. As we mentioned above TodoInheritedData contains only final fields and should never be modified. Instead, a new TodoInheritedData element should be created in the TodoProvider build method with the modified data.

In the to TodoProvider.dart a function called *onChangeFilter* is added. This function takes the new filter values as parameter and changes the value of the filter in the stateful widget calling *setState*. Doing so the *build* function is called again with the new filter value and a new TodoInheritedData widget is created.

void onChangeFilter(VisibilityFilter filter) {  
 setState(() {  
 this.filter = filter;  
 });  
}

The *onChangeFilter* function must be provided to the TodoInheritedData to make it accessible in the widget’s subtree. To do so a new parameter is added in the TodoInheritedData as follow.

@override  
Widget build(BuildContext context) {  
 return TodoInheritedData(  
 todos: todos,  
 onChangeFilter: onChangeFilter,  
 filter: filter,  
 child: widget.child,  
 );  
}

class TodoInheritedData extends InheritedModel<int> {  
 {...}  
 final void Function(VisibilityFilter) onChangeFilter;  
 {...}

Now that the *onChangeFilter* function is accessible down in the tree it can be called in the *onChange* function we provide inside the VisibilityFilterComponent DropdownButton.

onChanged: (filter) {  
 TodoInheritedData.of(context).onChangeFilter(filter!);  
},

The *filteredTodos* list can now be changed applying different filters. However, the Checkbox inside every TodoItem is just showing if the particular todo is completed or pending but its *onChange* function is still empty and does nothing when tapped. When a tap on the checkbox occurs a change in the corresponding Todo’s completed field should be fired and a rebuild of the TodoItems performed. (for the moment we don’t care if the TodoItem only or the entire TodoView is rebuilt). To do so TodoIhneritedData should provide also a function down the tree that allow to perform this change. Going back again to the TodoProvider.dart file a *onSetCompleted* function is added to the TodoProvider stateful widget. This function takes as parameter the id of the Todo to be changed and the new value for the completed field.

void onSetCompleted(int id, bool completed) {  
 assert(todoExists(id) != null, 'No todo with id : $id');  
  
 setState(() {  
 todos = todos.map((e) {  
 if (e.id == id) {  
 return Todo(  
 id: id,  
 name: e.name,  
 description: e.description,  
 completed: completed);  
 } else {  
 return e;  
 }  
 }).toList();  
 });  
}

The *todos* list is scanned using a map. Once the todo with the corresponding id is found its *completed* value is changed to the new value. Calling the *setState* method on the TodoProvider stateful widget will cause the build method to run again and to create another TodoInheritedData. All the elements of the TodoImheritedData subtree are rebuilt too.

At this point is possible to visualize the *filteredTodos* list, change the *filter* and update Todo’s *completed* field. To implement the *stats* tab the Stats component must be connected to the corresponding data and the TabSelector’s logic defined. First, the stats value is retrieved in the Stats component widget using the *of* method and visualized in the Ui.

class Stats extends StatelessWidget {  
 const Stats({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Stats");  
  
 return Text(TodoInheritedData.*of*(context).stats.toString());  
 }  
}

Then a new TabState variable is created in the HomePage called *tab* and set as TabState.todos by default. A function called *onTabChange* will call the *setState* method modifying the *tab* value and causing the build method to run again.

TabState tab = TabState.todos;  
  
void onTabChange(int index) {  
 setState(() {  
 tab = TabState.values.elementAt(index);  
 });  
}

*tab* value and *onTabChange* function are now passed to the TabSelector component as parameters and used to populate the BottomNavigationBar widget.

class TabSelector extends StatelessWidget {  
 final TabState currTab;  
 final Function(int) onTabChange;  
  
 const TabSelector(  
 {Key? key, required this.currTab, required this.onTabChange})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Tab Selector");  
  
 return BottomNavigationBar(  
 currentIndex: TabState.values.indexOf(currTab),  
 onTap: onTabChange,  
 items: TabState.values  
 .map((tab) => BottomNavigationBarItem(  
 label: describeEnum(tab),  
 icon: Icon(  
 tab == TabState.todos ? Icons.*list* : Icons.*show\_chart*,  
 ),  
 ))  
 .toList(),  
 );  
 }  
}

At this point all the basic functionalities have been implemented.

**Time spent: 2-3 hours**

**Lines of code written/updated: 86**

**Classes/widget created: 2** ( TodoInheritedData class and TodoProvider widget)

#### **Features addition**

**Todo creation feature:**

First thing is to create and make the *addTodo* feature/function accessible down the tree. A new function must be implemented in the TodoProvider widget and passed to the TodoInheritedData widget. This new function will be called *onAddTodo* and will take two parameters (*name* and *description*).

void onAddTodo(String name, String desc) {  
 Random rand = Random();  
 List<int> ids = todos.map((e) => e.id).toList();  
 int newId = rand.nextInt(1000) + 2;  
 while (ids.contains(newId)) {  
 newId = rand.nextInt(1000) + 2;  
 }  
 Todo newTodo = Todo(  
 id: newId,  
 name: name,  
 description: desc+ " " + newId.toString(),  
 completed: false);  
 List<Todo> newList = List.from(todos);  
 newList.add(newTodo);  
 setState(() {  
 todos = newList;  
 });  
}

After generating a new unique id it creates a new Todo object called *newTodo* with the *completed* field set to false. Adding the new Todo to the TodoProvider’s state *todos* list requires a bit of workaround. The state of a stateful widget is immutable. It can only be changed by the *setState* method. Unfortunately, the method *add* for lists is of type void and do not return a new list but instead add the new value to the existing one. For this reason directly calling the add method to the TodoProvider’s local lists *todos* will have no effect. That list is immutable and cannot be changed.

TodoProvider’s *todos* list must be completely replaced with a new list containing also the new todo. First a new temporary list called *newList* is created and populated with the element present in the *todos* list. Then the *newTodo* is added to this *newList* list. At this point is sufficient to replace the *todos* list with the new one inside the *setState* method.

To make this new function accessible down the tree is sufficient to add a new field in the TodoInheritedData (called *onAddTodo*) widget and pass the function on creation.

class TodoInheritedData extends InheritedWidget {  
 {...}  
 final void Function(String,String) onAddTodo;

{...}

@override  
Widget build(BuildContext context) {  
 return TodoInheritedData(  
 todos: todos,  
 onChangeFilter: onChangeFilter,  
 onAddTodo: onAddTodo,  
 onSetCompleted: onSetCompleted,  
 filter: filter,  
 child: widget.child,  
 );  
}

In the AddTodoPage a TextButton has been already set up and is ready to call this function once tapped. However, there is a small inconvenient. The AddTodoPage is accessed by pushing on top of the HomePage another route. In this new scope the Scaffold widget inside the AddTodoPage become the root of the tree of the current route. In other words, the AddTodoPage is not a part of the subtree of the HomePage but is a standalone tree instead. There is no instance of TodoProvider as ancestor of the AddTodoPage Scaffold widget and so it is not possible to call the *of* as before. Indeed calling the *of* method in a context where a TodoProvider is not present will cause the line

assert(result != null, 'No TodoInheritedData found in context');

to return *false* and rise a runtime error. The easiest method to proceed is to pass the *onAddTodo* function as a parameter to the AddTodoPage when we push it on top of the HomePage. So a new parameter called *addTodoCallback* is added to the AddTodoPage

class AddTodoPage extends StatefulWidget {  
  
 final void Function(String,String) addTodoCallback;

{. . .}

And the material app is notified about the necessity of this new argument in the AddTodoPage creation.

routes: {  
{. . .}

"/addTodo": (context) => AddTodoPage(  
 addTodoCallback: ModalRoute.*of*(context)!.settings.arguments  
 as Function(String, String)),  
},

At this point the *onChange* function of the TextButton inside the AddTodoPage can finally be populated as follow

TextButton(onPressed: () {  
 widget.addTodoCallback(textControllerName.text,textControllerDesc.text);  
 Navigator.*pop*(context);  
}

The current route (AddTodoPage) is also popped after the todo creation, and the HomePage is rebuilt (by the fact the TodoInheritedData changed).

**Time spent: 20-30 minutes**

**Lines of code written/updated: 24**

**Classes/widget created: 0**

**Todo updating feature:**

First thing is to create and make the *onUpdateTodo* feature/function accessible down the tree. A new function must be implemented in the TodoProvider widget and passed to the TodoInheritedData widget

void onUpdateTodo(int id, String newName,String newDesc) {  
 assert(todoExists(id) != null, 'No todo with id : $id');  
 List<Todo> newTodosList = todos.map((element) {  
 if (element.id == id) {  
 return Todo(  
 completed: element.completed,  
 description: newDesc,  
 name: newName,  
 id: element.id);  
 } else {  
 return element;  
 }  
 }).toList();  
 setState(() {  
 todos = newTodosList;  
 });  
}

class TodoInheritedData extends InheritedWidget {  
 ...  
 final void Function(int, String,String) onUpdateTodo;  
 ...

@override  
Widget build(BuildContext context) {  
 return TodoInheritedData(  
 todos: todos,  
 onChangeFilter: onChangeFilter,  
 onAddTodo: onAddTodo,  
 onSetCompleted: onSetCompleted,  
 onUpdateTodo: onUpdateTodo,  
 filter: filter,  
 child: widget.child,  
 );  
}

this new function will be called *onUpdateTodo* and takes three arguments: the *id* of the todo to be updated, the *newName* that should be set and the *newDesc*.

It first checks if a todo matching the id exists. Then, for the same immutability concept we dealt with when we spoke about the *onAddTodo* feature, a *newTodosList* is created and populated with the elements inside the *todos* list. Moreover, the todo with the corresponding id is update with the new name and new description. Finally, the *todos* list in the TodoProvider stateful widget is overridden with the *newTodosList* using the *setState* method.

For the same problem faced during the implementation of the add-Todo feature also in this case the *onUpdateTodo* function must be passed to the new route (no TodoProvider present in this context) as parameter. A new variable is added to the UpdateTodoPage ,beside the already existent one, called *callback*. This new variable will be a Function taking two Strings as arguments (the id will be already set up by the calling page).

class UpdateTodoPage extends StatefulWidget {  
 final Todo todo;  
 final void Function(String,String) callback;

The next step is to wrap the Row widget in the todo\_item.dart file inside a InkWell widget to make it sensible to taps.

InkWell(  
 onTap: () {},  
 child: Row...

Inside the onTap function the route UpdateTodoPage will be pushed but first a container for arguments must be set up. Indeed, Flutter Navigator allows to pass only a single object as argument between routes. In this case not only the *onUpdate Function* must be passed to the new route but also some information about the Todo itself. For this reason a wrapper class is created with the name *UpdateTodoPageArguments* as shown here

class UpdateTodoPageArguments {  
 final Todo todo;  
 final void Function(String ,String) updateState;  
  
 UpdateTodoPageArguments({required this.todo, required this.updateState});  
}

and inside the InkWell’s *onTap* function will be used to create a container for the arguments like this

Navigator.*pushNamed*(context, "/updateTodo",  
 arguments: UpdateTodoPageArguments(  
 todo: todo,  
 updateState: (String newName,String newDesc) {  
 TodoInheritedData.*of*(context, aspect: 0)  
 .onUpdateTodo(todo.id, newName,newDesc);  
 }));

A further change must be done in the MaterialApp’s “*/updateTodo”* route to populate the field of the UpdateTodoPage correctly.

routes: {  
 "/": (context) => const HomePage(),  
 "/updateTodo": (context) => UpdateTodoPage(  
 todo: (ModalRoute.*of*(context)!.settings.arguments  
 as UpdateTodoPageArguments)  
 .todo,  
 callback: (ModalRoute.*of*(context)!.settings.arguments  
 as UpdateTodoPageArguments)  
 .updateState,  
 ),

Now that the *onUpdateTodo* function is set up and correctly passed to the UpdateTodoPage is the time to call it inside the TextButton *onPressed* field like this

TextButton(onPressed: () {  
  
 widget.callback(textControllerName.text,textControllerDesc.text);  
 Navigator.*pop*(context);  
},

Once pressed the UpdateTodoPage will be popped, and the HomePage rebuilt to show the actual changes in the *todos* list.

**Time spent: 20-30 minutes**

**Lines of code written/updated: 43**

**Classes/widget created: 1 for arguments between routes**

#### **Render optimizations**

This was a pretty hard task. I spent some hour trying to figure out how make , when a single todo update occurs, rebuild the TodoItem only instead of the entire TodoView. Then I realized that it was just not feasible using InheritedWidgets. InheritedWidget indeed do not offer this possibility at all. Every widget in the TodoProvider’s subtree that access the state is registered as listener for state changes and once a state change occurs there are only two possibilities: notify all those widgets and rebuild them or not. In other words when a state change occurs and must be visualized the entire TodoProvider’s subtree must be rebuilt unconditionally. Flutter framework however offers a particular widget called InheritedModel to handle this scenario. InheritedModel work as InheritedWidget except for the fact that when a widget access the state (calling the *of* method) it must provide also a new additional parameter called *aspect*. *Aspect* can be whatever object for example a String or a Int but also a more complex data structure. The *aspect* parameter identifies on which part (or parts) of the state the widget is registering to.

**Migration to InheritedModel**

First thing to do is to substitute the extension to InheritedWidget with InheritedModel in the TodoInheritedData class (in the todo\_provider.dart file).

class TodoInheritedData extends InheritedWidget {

to

class TodoInheritedData extends InheritedModel<int> {

I decided to use Ints to identify aspects. In particular, widgets that need to rebuild on *filteredTodos* list structure change will register to aspect identified with the number **0**. Widgets that do never need to rebuild will register to aspect identified with number **1**. Widgets that need to rebuild when a change in a specific Todo with id **n** occurs will register to the aspect identified with the number **n**. (no Todos will have id with value 0 or 1. This is a convention I used to keep things simple. Other more complex structure could be used to avoid this behaviour). With *filteredTodos* structure I mean the length of the list. TodoView indeed should be entirely rebuilt only when a Todo is added or removed from the list changing its length. No todos replacement is considered by the fact that a replacement should be split into two separated actions ; a deletion and an insertion.

At this point the method *of* should be updated taking into account also the *aspect* parameter. Morevover, the *result* variable should be populated with the *inheritedFrom* static method belonging to the InheritedModel class instead of the *dependOnInheritedWidgetOfExactType* method belonging to InheritedWidget class.

static TodoInheritedData *of*(BuildContext context, {required int aspect}) {  
 final TodoInheritedData? result =  
 InheritedModel.*inheritFrom*<TodoInheritedData>(context, aspect: aspect);  
 assert(result != null, 'No TodoInheritedData found in context');  
 return result!;  
}

at this point all the lines of code that access the state with the *of* method must be changed taking into account the new implementation and the new *aspect* argument in this way

TodoInheritedData.*of*(context, aspect: aspect)

In particular the TodoView widget will pass as *aspect* the number 0 declaring that should be notified (and rebuild) only when a *filteredTodos*’s structure change occurs.

Instead TodoItem widgets will pass the corresponding Todo’s *id* as *aspect* parameter.

Now that every widget is registered only to the desired aspect of the data, is necessary to “teach” the TodoInheritedData to recognize which aspect of the data actually changed when a state change occurs. To do so InheritedModel provides a method called *updateShouldNotiyDepenedent* that is just like the InheritedWidget’s one *updateShouldNotify* but this time takes as argument also a Set of ints called *dependencies* (aspects). This method is called once for every widget that registered to state changes and the dependencies variable will contains all aspects the widgets registered to (only one for widget in our case). As follow the implementation of the method:

@override  
bool updateShouldNotifyDependent(  
 TodoInheritedData oldWidget, Set<int> dependencies) {  
 int currLen = filteredTodos.length;  
 int prevLen = oldWidget.filteredTodos.length;  
 bool structureRebuildlen = (dependencies.contains(0) && currLen != prevLen);  
 if (structureRebuildlen == true) {  
 return true;  
 } else {  
 List<int> currIds = filteredTodos.map((todo) => todo.id).toList();  
 List<int> prevIds =  
 oldWidget.filteredTodos.map((todo) => todo.id).toList();  
 bool sameIds = listEquals(currIds, prevIds);  
 bool structureRebuildcomp = (dependencies.contains(0) && !sameIds);  
 if (structureRebuildcomp == true) {  
 return true;  
 } else {  
 List<bool> components = [];  
 for (var element in filteredTodos) {  
 components.add(dependencies.contains(element.id) &&  
 !oldWidget.filteredTodos.contains(element));  
 }  
 bool res = components.fold(false,  
 (bool previousValue, bool element) => previousValue || element);  
 return res;  
 }  
 }  
}

This was tough to code but in the end worked well for the purpose. The figure 2.2.2 presents the pseudocode for the method

If( widgetRegisteredForStructureChange && strucutureChangeOccured)

{ return true;}else{

If( widgetRegisteredForSpecificTodoChange && thatTodoChanged){

Return true;}else{return false;}

Once the TodoItem’s checkbox is tapped just the TodoItem is rebuilt. No visual changes are shown, however. The widget will rebuild with the same information as before and this is due to the fact that the build method refers to the local Todo variable. This variable is populated on the TodoItem creation and cannot be changed. Indeed, a Todo is passed as argument in the constructor method from the TodoView and from that moment on will remain the same. No visual changes are shown because this local Todo indeed did not change. It is a copy of the actual Todo present in the *filteredTodos* list and for this reason is not affected by changes. This is a really bad behavior and is cause by the fact that sometimes, during programming , more than one level of information caching is required/used to avoid effort in coding or performance issues. In other words, a local copy of the data kept and referred to in case of data access in order to optimize the accesses in the main storage that can become quite expensive in large scenarios. A great example of that is the local copy of the database’s data used in many applications. Is more effective to fetch data from the database, save them locally, manipulate this local copy and only in case of real necessity access again the database to store them or retrieve other data. In large applications (but also in small ones like in this cases) more than one level of data caching is used. Particular attention is required to handle those levels to avoid inconsistency in what is visualized and the real data. In this case the *filteredTodos* list actually changed but the UI did not reflected it. The problem was generate by the fact that a copy of the real Todo was passed to the TodoItem widget instead of the id of the Todo and then use it to look up for the Todo in the centralized state (the TodoInheritedData). This of course will require more computational effort but also will guarantee a lot more stability and robustness.

Saying that the TodoItem’s local variable Todo is replaced with a int that represents the id of the Todo that the widget is visualizing. Then in the *build* method the corresponding Todo is looked up.

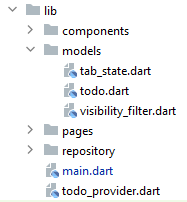
class TodoItem extends StatelessWidget {  
 final int id;  
  
 const TodoItem({Key? key, required this.id}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 final Todo todo = TodoInheritedData.*of*(context, aspect: id)  
 .todos  
 .where((element) => element.id == id)  
 .first;

At this point the application is working as intentioned and the renders optimization was successfully accomplished.

**Time spent: 8-10 hours**

**Lines of code written/updated: 49**

**Classes/widget created: 0**

****

### Redux implementation

In questa sezione la gestione dello stato verrà gestita tramite Redux.

**\paragraph**{Base funtionalities} \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**

**\subparagraph**{The AppState}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

Being the state centralized when the redux solution is used, another model must be defined to connect all the components in a unique class. Subparts of the state have been already modelled in the implementation of the shared parts of the application RIFERIMENTO. They are used now to compose a new class, called AppState, that will group all the subparts in a unique place throughout the entire application. The list of filtered todos will not be part of the AppState. In Redux the centralized state is kept as simple as possible and the parts of the state that can be computed or derived should not take part in it. There are two possibilities basically; compute every time the filtered list on the visualization side or use Selectors. This aspect will be deeper investigated later but for now lets start to define the three main reducers.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

class AppState {  
  
 VisibilityFilter visibilityFilter;  
 List<Todo> todos;  
 TabState tabState;  
  
 AppState({this.todos=const [],this.tabState= TabState.todos,this.visibilityFilter= VisibilityFilter.all});  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{The actions}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The application’s state is basically composed by model classes. In order to mutate the state is necessary to define actions. Actions will then be processed by reducers to change the state. Actions are simple classes. We start defining the actions needed to change the state regarding the list of todos. The list of todos is represented by a object of type list in the AppState. Actions can generate a state transition or none but cannot generate more than one state emission. For this reason two actions are needed to define the todos fetching. One action to indicate the start of the fetching process and one action to actually insert the fetched todos in the todos list in the AppState. The first action will be called LoadTodoAction. The second one will be called LoadTodoSucceededAction and will contain the list of fetched todos.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

class LoadTodoAction{  
 @override  
 String toString(){  
 return "loadTodoAction";  
 }  
}  
class LoadTodoSucceededAction{  
 List<Todo> todos;  
 LoadTodoSucceededAction(this.todos);  
}

\end{minted}

\mbox{}

\end{code}

Moreover , an action to set the completed field of a specific todo is required to be used in the TodoItem checkbox. This action will be called SetCompletedAction and will contain the id of the todo to be changed and the new value for the completed field.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

class SetCompletedTodoAction {  
 final int id;  
 final bool completed;  
  
 SetCompletedTodoAction(this.id, this.completed);  
}

\end{minted}

\mbox{}

\end{code}

Only two more action are needed to handle the tab and the visibility filter changes. They will be called respectively SetTabAction and SetVisibilityFilterAction. They brings the new tab and the new filter to be set.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class SetTabAction{  
 final TabState newtab;  
  
 SetTabAction(this.newtab);  
}

class SetVisibilityFilterAction{  
 VisibilityFilter filter;

SetVisibilityFilterAction(this.filter);  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{Reducers}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

Defining the state model and the actions was very simple. Is the turn now for reducers which will link the state with the actions. Although the usage of the Redux state management solution requires to centralize all the state in a unique component doesn’t mean that the state’s logic(?) cannot be split in subparts. It is like a tree with a single root where the root is represented by the app state’s reducer. It can split up, however, in many sub-reducers and every one of them can split up furthermore into other sub-reducers. So, in the end, the state is segmented and stored in single place, but its pieces can still be managed separately and independently. During the reasoning(?) process the whole state is broken into small pieces step by step using a top-down approach however in the implementation process a bottom-up approach is usually used. This is due to the fact that, during the implementation process , smaller bricks are needed to build bigger components. In this presentation, for example, is not possible to define directly the AppState reducer even if, logically, it should be the first one to be implemented. The AppState reducer will be composed by multiple sub reducers, in our case three. There will be a reducer for the list of todos, a reducers for the filter and a reducer for the tab.

**\subparagraph**{The todoReducer}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The required functionalities are the loading of the todos from the db and the setting of the completed field of a specific todo. As said before, reducers are basically functions. They take the previous state and an action and return the next state. Speaking about the list of todos , the state is an object of type List<Todo>. It initially empty and is filled up once an event of type loadTodoSucceededAction is received. The list of fetched todos is found inside the action and used to return the new state for the list of todos. The reducer in this case just takes the list and returns it.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

List<Todo> \_setLoadedTodo(List<Todo> todos, LoadTodoSucceededAction action) {  
 return action.todos;  
}

\end{minted}

\mbox{}

\end{code}

Another reducer is needed to handle the action of setting the completed field of a specific todos. In the case the received action will be of type SetCompletedTodoAction. The todo matching the id contained in the action is searched and updated using the list’s map method and then a new instance of the list is created and returned. The necessity to create another instance comes from the fact that the transition would not be recognised in case the state’s list is just updated and not substituted.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

List<Todo> \_setCompletedTodo(List<Todo> todos, SetCompletedTodoAction action) {  
 List<Todo> newList= todos.map((todo) => todo.id == action.id  
 ? Todo(  
 id: action.id,  
 name: todo.name,  
 description: todo.description,  
 completed: action.completed)  
 : todo).toList();  
  
 return List.from(newList);  
}

\end{minted}

\mbox{}

\end{code}

Now that the two reducers for the action previously created has been implemented we will usesome tools the redux package provides in order to combine them in a unique reducer that binds the received action to the correct sub-reducer. These tools are the combineReducers function and the TypedReducer class. The TypedReducer class is, indeed, a typed class that helps to avoid nested if-else structure which can generate lot of boilerplate and code unreadability. It binds a specific action to a reducer. combineReducers , instead, is a function that creates a reducer composing sub-reducers provided in the form of TypedReducers. The two previously defined reducers are now merged to an unique one called todosReducer.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

final todoReducer = combineReducers<List<Todo>>([  
 TypedReducer<List<Todo>, LoadTodoSucceededAction>(\_setLoadedTodo),  
 TypedReducer<List<Todo>, SetCompletedTodoAction>(\_setCompletedTodo),

]);

\end{minted}

\mbox{}

\end{code}

the alternative would have been to write the todosReducer like presented RIERIEMENTO. The output is the same but the code is clearly more readable.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

**final** todosReducer = (List<Todo> todos, action) {

**if** (action **is** LoadTodoSucceededAction) {

**return \_setLoadedTodo(todos,action)**;

} **else** **if** (action **is** SetCompletedTodoAction) {

**return** \_setCompletedTodo(todos, action);

} **else** {

**return** state;

}

};

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **The tab Reducer and the visibility Filter reducer**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The process is the same as before. Two reducers ,called\_setTabState and \_setVisibilityFilter ,are created to handle respectively actions of type SetTabAction and SetVisibilityFilter action. In both cases the value contained in the action is used to produce a new state. Both of them are used to create other two reducers called tabReducer and visibilityFilterReducer using the combineReducers function introduced earlier. In this case the usage of the combineReducers function wasn’t really necessary by the fact that it combines only one reducer. However in case new actions may be introduced it will get handy.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
TabState \_setTabState(TabState state, SetTabAction action){  
 return action.newtab;  
}

VisibilityFilter \_setVisibilityFilter(  
 VisibilityFilter oldState, SetVisibilityFilterAction action) {  
 return action.filter;  
}

final tabReducer= combineReducers<TabState>([  
 TypedReducer<TabState, SetTabAction>(\_setTabState),  
  
]);

final visibilityFilterReducer = combineReducers<VisibilityFilter>([  
 TypedReducer<VisibilityFilter, SetVisibilityFilterAction>(  
 \_setVisibilityFilter)  
]);

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **The AppState reducer**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

Finally all the basic bricks have been implemented and can be used now to create the biggest reducer; the AppState reducer. It is of course a function that takes the current AppState and an action. It then compose a new AppState instance using the sub-reducers just created and passing to them the received action. Every sub-reducer processes the action to understand if it needs to perfom a transition in the handled part of the state. And that’s it, a unique component handling all the application’s state has been defined combining more sub parts.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

AppState appStateReducer(AppState appState, action) {  
 return AppState(  
 todos: todoReducer(appState.todos, action),  
 tabState: tabReducer(appState.tabState, action),  
 visibilityFilter: visibilityFilterReducer(appState.visibilityFilter,action));  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **The Middleware**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

All we need to start accessing the state in the UI is settled up but the way in which the todos are fetched from the database is not really clear yet. Two actions were set up for this purpose but only has been handled by a reducer for the moment. The fact is that the process of fetching todos from the database is not immediate. It is asynchronous with respect to the application workflow. Reducers are not suited for handling asynchronous code , they need to be pure and as simple as possible. There are many ways in practice to deal with asyn code but the most “correct” one is to use middlewares. Middlewares has been introduce HERE RIEFERIMENTO. The act as a sort of proxy from the action dispatch and the reducers. One or more middleware can be provided to be executed on a action call. They are used to handle asynchronous code but also action that generate side effects. In our case we defined an action called LoadTodoAction that do not have any meaning in terms of reducers. Once dispatched it will pass through one or more middleware before reaching the reducers. A middleware is just a function that takes three parameters: the store, an action and the next dispatcher. It processes the action , probably accessing the store, and the pass the action to the next middleware. In our case a new middleware function is defined and called loadTodoMiddleware. It just check is the dispatched action if of type LoadTodoAction and in case it is starts to load the todos from the database. Once the fetching process is completed it dispatches another action of type LoadTodoSuccededAction that , this time will be handled, by the AppState reducer.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

void loadTodosMiddleware(Store<AppState> store, action, NextDispatcher next) {  
  
 if (action is LoadTodoAction) {  
 TodoRepository.*loadTodos*().then((todos) {store.dispatch(LoadTodoSucceededAction(todos));} );  
 }  
 next(action);  
}

\end{minted}

\mbox{}

\end{code}

All the ingredients are now available to start composing our UI.

**\subparagraph**{ **Making the state accessible**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

Like almost every solution used in this overview , also redux uses providers to make the state accessible down the widgets tree. The state is unique and tough should obviously be placed in the root of the widgets tree or some part of the tree would not be covered by its accessibility. To do so a StoreProvider is used to wrap the entire app positioned right below the MyApp widget. An object of type Store must be provided to the StoreProvider widget. The typed class Store is provided by the redux package. Store class is a typed class meaning that the type of the global state to be handled must be provided. In our case the global state is of the type AppState defined above. The Store class constructor takes three parameters. The reducer for the AppState , an instance of an object of type AppState that identifies the initial state and a list of middlewares. In our case a single middleware is passed. In order to fetch the todos on the application start we will use the init method of the stateful widgets. The HomePage widget is already stateful and so its init method is used to dispatch the first action : the LoadTodosAction. For simplicity the function to be executed in the init method is passed from the MyApp widget where the store is already available to the HomePage using a newly created parameter.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

void main() {  
  
  
 WidgetsFlutterBinding.*ensureInitialized*();  
  
 runApp(MyApp(  
 store: Store<AppState>(appStateReducer,  
 initialState: AppState(), middleware: [loadTodosMiddleware])));  
  
  
}

class MyApp extends StatelessWidget {  
 final Store<AppState> store;  
  
 const MyApp({Key? key, required this.store}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Material App");  
 return StoreProvider(  
 store: store,  
 child: MaterialApp(initialRoute: "/",  
 routes: {  
 "/": (context) => HomePage(onInit: (){  
 store.dispatch(LoadTodoAction());  
 },),  
 "/addTodo":(context) =>const AddTodoPage(),  
 "/updateTodo" : (context)=> UpdateTodoPage(todo: (ModalRoute.*of*(context)?.settings.arguments as Todo))  
 },  
 ),  
 );  
  
 }  
}

\end{minted}

\mbox{}

\end{code}

Inside the HomePage widget the initState method is defined and the store is accessed for the first time. To access the store a StoreConnector widget is used. Its task is to access the state and to create a viewmodel containing the information related to the pages usage only. The viewmodel is really important because idenfies the prospective from which the widget is looking at the state. If the viewmodel changes the widget is rebuilt. Not always a state change produces a viewmodel change. The StoreConnector widget takes two types in its definition. The global state’s type,the AppState, and the type of the viewmodel it is going to compose, in our case just a TabState object. The HomePage ,indeed, only need the part of the state related to the tab and so it is not necessary to create an ad hoc view model. The model of the TabState defined hereRIFERIMENTO is enough to contain the interesting part of the state. Two StoreConnector’s fields must be filled to make it work correctly. The converter field and the builder field. In the first one a function must be provided that manipulate the state to return an object of the type previously specified, in our case a object of type TabState. The builder field is populated with a function that returns a widget. Inside this function the current context and the object returned from the converter function are available. The current tab value is used to populate the Scaffold widget as usual.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

@override  
void initState() {  
 widget.onInit();  
 super.initState();  
}

@override  
Widget build(BuildContext context) {  
 print("Building HomePage");  
  
 return StoreConnector<AppState, TabState>(  
 converter: (store) => store.state.tabState,  
 builder: (context, currTab) {  
 return Scaffold(  
 appBar: AppBar(  
 title: const Text("Todo App"),  
 actions: const [VisibilityFilterComponent()],  
 ),  
 body: currTab == TabState.todos ? const TodoView() : const Stats(),  
 bottomNavigationBar: const TabSelector(),  
 floatingActionButton: currTab == TabState.todos  
 ? FloatingActionButton(  
 child: const Icon(Icons.*plus\_one*),  
 onPressed: () {  
 Navigator.*pushNamed*(context, "/addTodo");  
 })  
 : Container());  
 },  
 );  
}

\end{minted}

\mbox{}

\end{code}

We could now start to implement the component widgets using the AppState but first a short digression about Selectors must be taken.

**\subparagraph**{ **Selectors**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

They are introduced hereRIFERIMENTO . Selectors are used to compute those parts of the state that entirely depends on other parts. In our case the filtered list is not included in the AppState , also because would be hard to deal with it in the centralized state syncornizing it with the changes in the todos list and the filter. The easiest way to deal with the filtered list is to compute it in the UI layer before buildeing the widget, in our case the TodoView widget. However, this way of doing can become quite heavy soon by the fact that the computation of the filtered list is perfomed every time the widge tis rebuilt and it can be costly sometimes. For this reason Selector are used to memoize the previously computed value and reuse them in case they did not change. Selectors are just functions that take as input the state an return an object composed with it. All the memorization part is perfomed by a third party package called Reselect. In order to use it it must be included in the pubspec.yaml file under the dependencies field. It provides some functions with the name createSelector that take care of memoize the values and understand when to recompute them. Selectors can be simple o composed. For example two simple selectors can be implemented in our case. One that takes the state and return the filter and on that takes the state and return the list of todos. Selectors can be composed with other selector to create articulated objects. Selectors are composed using the createSelector function followed by a number. For example a selector computing the list of completed todos can be built using the todos selctor we just created and the same for computing the list of pending todos. A Selector is created then to compute the filtered list. It will use four other selectors. The list of todos selector, the filter selector, the list of pending todos selector and the list of completed todos selector outputs are composed to compute the filtered list of todos. Besides of making the code more readable , selectors also optimize the application performances.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

final todosSelector = (AppState state) => state.todos;  
  
final filterSelector = (AppState state) => state.visibilityFilter;  
  
final completedTodosSelector = createSelector1(  
 todosSelector,  
 (List<Todo> todos) =>  
 todos.where((todo) => todo.completed == true).toList());

final pendingTodosSelector = createSelector1(  
 todosSelector,  
 (List<Todo> todos) =>  
 todos.where((todo) => todo.completed == false).toList());  
  
final filteredTodosSelector = createSelector4(  
 todosSelector, filterSelector, completedTodosSelector, pendingTodosSelector,  
 (List<Todo> todos, VisibilityFilter filter, List<Todo> completed,  
 List<Todo> pending) {  
 switch (filter) {  
 case VisibilityFilter.completed:  
 return completed;  
 case VisibilityFilter.notCompleted:  
 return pending;  
 case VisibilityFilter.all:  
 return todos;  
 }  
});

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **The TodoView component**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

After this short digression about selectors we are ready now to set up the TodoView component. The ListView widget is wrapped into a StoreConnector widget type with the AppState and a list of todos. Indeed, the TodoView component just needs to access the part of the state concerning the filtered list of todos that is in the end an object of type List<Todo>. However, the list of filtered todos is not directly available in the AppState but the selectors we just defined can to be used to compute the filtered list. In the StoreConnector converter field a function is provided that takes the store and returns the filtered list using the filteredTodosSelector function. The converter function output is then available inside the builder field function and we can use it to populate the ListView widget as usual.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class TodoView extends StatelessWidget {  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return StoreConnector<AppState, List<Todo> >(  
 distinct: true,  
 builder: (context, todos) {  
 print("Building TodoView");  
 return ListView.builder(  
 itemCount: todos.length,  
 itemBuilder: (context, index) {  
 return TodoItem(  
 todo: todos.elementAt(index),  
 );  
 },  
 );  
 },  
 converter: (store) {  
 return filteredTodosSelector(store.state);  
 });  
 }  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **The TodoItem component**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The TodoItem component does not need any modification with respect to the implementation reposrted HERE RIFERIMENTO. Indeed, it just receives a todo from the parent widget and exposes it to the user. The only missing part in the onChanged field of the Checkbox that is left empty because its implementation changes between diffent solutions. Inside the onChage field a function must be provided that updates the completed field of the todo visualized by the TodoView widget. Providing that function is really easy, the store is accessed using the of method of the StoreProvider widget. ( the of method gets the nearest instance of the store of type provided in the call, in our case AppState)The store is then used to dispatch an action of type SetCompletedTodoAction created using the id of the visualized todo and the completed Boolean value provided by the onChange fuction.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

Checkbox(  
 value: todo.completed,  
 onChanged: (completed) {  
 StoreProvider.*of*<AppState>(context).dispatch(  
 SetCompletedTodoAction(todo.id, completed!));  
 }),

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{The VisibilityFilterSelector component}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The dfhdhfdhs component only accesses the part of the state concerning the filter. The entire DropdownButton widget is wrapped into a StoreConnector widget. The StoreConnector’s converter function will return a value of type VisibilityFilter. In the converter field a function taking the store and returning the filter value must be provided. To do so the filterSelector implemented earlier can be used or instead the filter value can be accessed directly by the store, both solution are equivalent. The ouput of the converter function can be accessed in the builder field function and used to populate the DropdownButton widget. The only part that differs from the implementation provided HERE RIFERIMENTO is the function used in the onChanged field of the DropdownMenuItem widgets. It gets an instance of the store using the StoreProvider widget’s of method and then dispatches an action of type SetVisibilityFilterAction using the DropdownMenuItem corresponding filter value.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
@override  
Widget build(BuildContext context) {  
 return StoreConnector<AppState, VisibilityFilter>(  
 converter: (store) => filterSelector(store.state),  
 builder: (context, activeFilter) {  
 print("Building Visibility filter");  
  
 return DropdownButton<VisibilityFilter>(  
 value: activeFilter,  
 items: VisibilityFilter.values.map((filter) {  
 return DropdownMenuItem<VisibilityFilter>(  
 child: Text(describeEnum(filter)), value: filter);  
 }).toList(),  
 onChanged: (filter) {  
 StoreProvider.*of*<AppState>(context)  
 .dispatch(SetVisibilityFilterAction(filter!));  
 },  
 );  
 },  
 );

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{The TabSelector component}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

Also in this case the TabSelector components needs to read and write the AppState. A StoreConnecgor widget is used to wrap the BottomNavigationBar widget making the state accessible to it. The view model to be outputted by the converter function is just an object of type TabState. An arrow function that returns the current AppState’s tab is provided to the converter field. The function output is then used in the builder field function to populate the BottomNavigationBar widget. To notice the function provided in the onTap field that accesses the AppState using the StoreProvider’s of method and dispatches a new action of type SetTabAction with the corresponding bottomNavigationBarItem’s teb value.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

class TabSelector extends StatelessWidget {  
 const TabSelector({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
  
 return StoreConnector<AppState, TabState>(  
 converter:(store)=>store.state.tabState,  
 builder: (context, currTab) {  
 print("Building Tab Selector");  
  
 return BottomNavigationBar(  
 currentIndex: TabState.values.indexOf(currTab),  
 onTap: (index)=>StoreProvider.of<AppState>(context).dispatch(SetTabAction(TabState.values.elementAt(index))),  
 items: TabState.values  
 .map((tab) => BottomNavigationBarItem(label: describeEnum(tab),  
 icon: Icon(  
 tab == TabState.todos ? Icons.list : Icons.show\_chart,  
 ),  
 ))  
 .toList(),  
 );  
 },  
 );  
 }  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{The Stats component}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The stats component only needs to read the state. The Center widget is wrapped into a StoreConnector widget. In the converter field function the completedTodoSelector function is used to access the list of completed todos and its length is then computed. In this case ,indeed, the viewmodel to be outputted is just an int value. The int value is then displayed in the Text widget.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

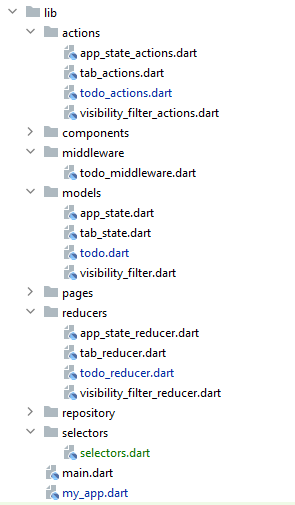
class Stats extends StatelessWidget {  
 const Stats({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return StoreConnector<AppState, int>(  
 builder: (context, completed) {  
 print("Building Stats");  
 return Center(child: Text(completed.toString()));  
 },  
 converter: (store) {  
 return completedTodosSelector(store.state).length;  
 },  
 );  
 }  
}

\end{minted}

\mbox{}

\end{code}

At this point all the base functionalities have been implemented and are working fine.



**\paragraph**{Features addition} \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**

As usual the first thing to do is to make the state provide a way of adding and updating todos. For this purpose two new action are created with the name AddTodoAction and UpdateTodoAction respectively. In the AddTodoAction the name and the description for the todo to be create are contained. In the UpdateTodoAction ,besides the new name and description for the todo, also the id of the todo to be modified is provided.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class AddTodoAction {  
 final String name;  
 final String desc;  
  
 AddTodoAction(this.name, this.desc);  
}  
  
class UpdateTodoAction{  
 final String name;  
 final String desc;  
 final int id;  
  
 UpdateTodoAction(this.name,this.desc,this.id);  
}

\end{minted}

\mbox{}

\end{code}

In order to handle these new actions two reducers must be created. They are called respectively addTodo and updateTodo and are just functions. They are inserted in the reducers of the list of todos. The addTodo reducer takes as argument the current list of todos and the AddTodoAction object. It first creates the new todo using the information contained in the action object and generating a new unique id. Then creates a completely new list and populates it with the old list elements and the new todo.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

List<Todo> \_addTodo(List<Todo> todos, AddTodoAction action) {  
 Random rand = Random();  
 List<int> ids = todos.map((todo) => todo.id).toList();  
 int newId = rand.nextInt(1000) + 2;  
 while (ids.contains(newId)) {  
 newId = rand.nextInt(1000) + 2;  
 }  
 Todo newTodo = Todo(  
 id: newId,  
 name: action.name,  
 description: action.desc + " " + newId.toString(),  
 completed: false);  
  
 return List.from(todos)..add(newTodo);  
}

\end{minted}

\mbox{}

\end{code}

The updateTodo reducer takes the current list of todo and an action of type UpdateTodoAction as arguments. It first modifies the todoof the current list of todos with the id matching the one contained in the action. Then it generates a new list with the element contained in the current one and returns it.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
List<Todo> \_updateTodo(List<Todo> todos, UpdateTodoAction action){  
  
 List<Todo> newList= todos.map((todo) => todo.id == action.id  
 ? Todo(  
 id: action.id,  
 name: action.name,  
 description: action.desc,  
 completed: todo.completed)  
 : todo).toList();  
  
 return List.from(newList);  
}

\end{minted}

\mbox{}

\end{code}

These new reducers are then combined with the other existing ones in the combineReducers function and linked with the corresponding action using the type class TypedReducer.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

final todoReducer = combineReducers<List<Todo>>([  
 TypedReducer<List<Todo>, AddTodoAction>(\_addTodo),  
 TypedReducer<List<Todo>, LoadTodoSucceededAction>(\_setLoadedTodo),  
 TypedReducer<List<Todo>, SetCompletedTodoAction>(\_setCompletedTodo),  
 TypedReducer<List<Todo>, UpdateTodoAction>(\_updateTodo),  
]);

\end{minted}

\mbox{}

\end{code}

And that’s it, the AppState can now handle actions of type AddTodoAction and UpdateTodoAction. These new functionalities can now be used in the AddTodoPage and in the UpdateTodoPage easilt due to the fact that they can be accesses directly. The StoreProvider widget has been , indeed, positioned in the root of the application to be available in all the sub trees. The onPressed field ‘s function of the TextButton widget in the AddTodoPage dispatches an Action of type AddTodoAction in the AppState utilizing the name and description’s textControlled text field.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

TextButton(  
 onPressed: () {  
 final AddTodoAction action= AddTodoAction(textControllerName.text, textControllerDesc.text);  
 StoreProvider.*of*<AppState>(context).dispatch(action);  
 Navigator.*pop*(context);  
 },  
 child: const Text("Create"))

\end{minted}

\mbox{}

\end{code}

And the same is done in the UpdateTodoPage dispatching an action of type UpdateTodoAction composed using the textControllers’s text values and the id of the todo passed as argument from the TodoItem widget.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

TextButton(  
 onPressed: () {  
 final UpdateTodoAction action= UpdateTodoAction(textControllerName.text,textControllerDesc.text,widget.todo.id);  
 StoreProvider.*of*<AppState>(context).dispatch(action);  
 Navigator.*pop*(context);  
 },  
 child: const Text("Confirm"))

\end{minted}

\mbox{}

\end{code}

All the features have been successfully added and work well.

**\paragraph**{Rendering optimization} \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**

In order to perform the optimizations, we will leverage on the fact that the StoreConnector widgets only rebuilds when the viewmodel ,it relies on , changes. Or better, this behavior happens when a specific field of the StoreConnector widget ,called distinct, is set to true. This feature ,the redux package provides, makes the optimization process really easy. We just need to define the correct viewmodel and the correct equality operator between viewmodels. Some changes must be done in the TodoView widget and in the TodoItem widget. Firstly, the TodoItem widget needs to interact with the state to understand when a change regarding its todo is performed. For the moment , indeed ,the TodoItem widget just receives a todo instance from the parent widget and visualizes it. As usual, we use a StoreConnector widget to read the state in the TodoItem widget. The complete instance of the todo to be visualized is not necessary to be passed by the parent anymore but the id only is enough. Using the id , the corresponding todo is searched in the store by the converter field’s function and passed to the builder field. The widget returned by the builder function remains the same with the only exception that it uses the todo instance got from the store instead of the one passed by the parent widget.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class TodoItem extends StatelessWidget {  
 final int id;  
  
 const TodoItem({Key? key, required this.id})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return StoreConnector<AppState, Todo>(  
 distinct: true,  
 converter: (store) =>  
 store.state.todos.firstWhere((element) => element.id == id),  
 builder: (context, todo) {  
 print("building: Todo Item $id ");  
  
 return InkWell(. . .);  
 });  
 }  
}

\end{minted}

\mbox{}

\end{code}

Now that the TodoItem widget watches for its own todo changes ,the optimization process regarding the TodoItem widget is automatically performed. We said, indeed, that the Storeconnector widget is rebuilt every time its viewmodel changes and we set the viewmodel as the corresponding todo instance in the AppStore. Once a state change occurs, the StoreConnector widget compares the current todo with the new one and, in case they differ, it rebuilds. To notice that this mechanism works because we redefined the equality operator between objects of type Todo HERE RIFERIMENTO. An object of type Todo is equal to another one if all their internal values match. This equality differs from the traditional one by the fact that do not check for the entity’s equality. Indeed, if it would, the two todos would appear to be different everytime. This because every time a new state is emitted the list of todos is recreated from scratch and also its internal todos are. In the dart language two different objects end to be different also if their internal values are exactly the same. This way of handling object already showed up numerous times in this overview and will show up even more in the next implementations. Going back to the to the TodoItem component , it already uses the correct equality operator to compare different viewmodels and so is capable to understand when to rebuild by itself. The same process must be done in the TodoView widget. It already uses a StoreConnector widget but in the converter function derives the list of filtered todos from the store. However, as we just said, two plain lists appear to be always different by default. Moreover, it is not possible to redefine the equality operator for list and in every case it would not be a good idea. A simple way to handle this scenario is to create ad hoc viewmodel and refine its equality operator in order to implement our own rebuilding logic. To do so a local class is created and called ViewModel. This local class just contains a list of todos.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class \_ViewModel {  
 final List<Todo> todos;  
  
 \_ViewModel({required this.todos});  
  
}

\end{minted}

\mbox{}

\end{code}

Inside the ViewModel class the equality operator is overridden making two ViewModel instances equal when their length matches and their internal ids match too. In this way the TodoView widget is rebuild only in case the filtered todos list has reported a structural change. (structural and not structural changes are exaplained HERE RIFERIMENTO).

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

@override  
 bool operator ==(Object other) {  
 return ((other is \_ViewModel) &&  
 todos.length == other.todos.length &&  
 todos.every(  
 (todo) => other.todos.any((element) => todo.id == element.id)));  
 }  
  
 @override  
 *// TODO: implement hashCode* int get hashCode => todos.hashCode;  
\end{minted}

\mbox{}

\end{code}

After setting the distinct field to true and modifying the converter function to return a ViewModel instead of a plain List the optimizations are basically done.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class TodoView extends StatelessWidget {  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return StoreConnector<AppState, \_ViewModel>(  
 distinct: true,  
 builder: (context, vm) {  
 print("Building TodoView");  
 return ListView.builder(  
 itemCount: vm.todos.length,  
 itemBuilder: (context, index) {  
 return TodoItem(  
 id: vm.todos.elementAt(index).id,  
 );  
 },  
 );  
 },  
 converter: (store) {  
 return \_ViewModel(todos: filteredTodosSelector(store.state));  
 });  
 }  
}

\end{minted}

\mbox{}

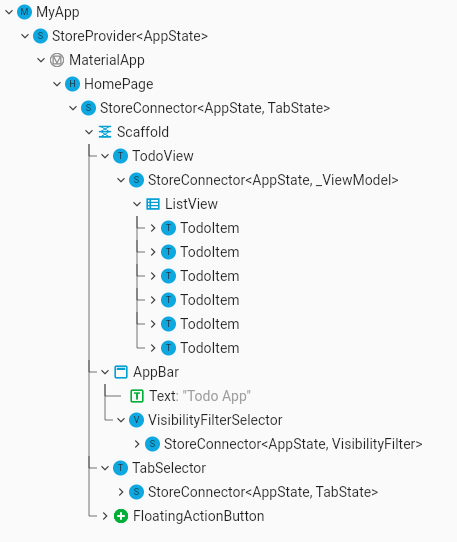
\end{code}

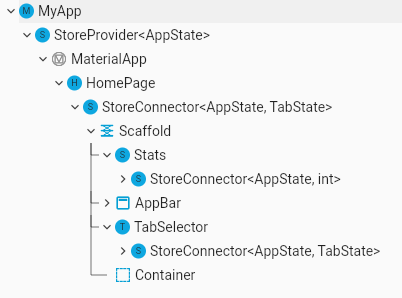
To put the  [the icing on the cake](https://context.reverso.net/traduzione/inglese-italiano/the+icing+on+the+cake) we set the distinct field to true also in the VisibilityFilterSelector compoenent and in the TabSelector component. In this way they are rebuilt only in case their prospective of the state changes and not at every transition.

**\paragraph**{Conclusions} \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**

Immagine che contiene testo

Descrizione generata automaticamente





### Implementazione con BLoC

In questa sezione verrà utilizzata la soluzione Bloc per la gestione dello stato.

#### Implementazione delle funzioni base

**\subparagraph**{States}\mbox{}\\

**\label{subpar:todo\_app\_inherited\_widget\_core\_state}**

The application state is decomposed in four smaller states: the state of the list of todos, the state of the filtered list of todos and the filter, the state of the statistic and the state of the tab. The state of the list of todos contains the whole list of todos. The state of the filtered list of todos and the filter contains a filter ,of type VisibilityFilter ,and a list of todos matching the filter value. The state of the stats contains an int number indicating the number of completed todos. Lastly, the state of the tab contains the value of the HomePage’s active tab. The state of the list of todos and the state of the tab are independent. The state of the filter and the state of the stats , instead, are directly linked to the state of the list of todos. They will , indeed, react to the changes in the state of the list of todos and update consequently.

**Lo stato dei todo**

Per prima cosa iniziamo a definire e a dare un nome ai possibili stati in cui si potrà trovare la lista dei todo. Questi ultimi saranno solo due : TodosLoadingState e TodosLoadedState. Lo stato Loading starà a indicare che la lista dei todo è ancora in caricamento. Lo stato Loaded invece indicherà che la lista dei todo è stata caricata del database ed e disponibile. Per definire questi tre stati creiamo prima una classe astratta chiamata TodosState. Essa dovrà ,inoltre, estendere la classe Equatable. La classe Equatable servirà in seguito per definire l’operatore di uguaglianza tra stati e poter comprendere correttamente quando un nuovo stato viene emesso. I due stati che andremo a definire, Loading e Loaded , estenderanno entrambi la classe astratta TodosState. Lo stato Loading non avrà altre informazioni al suo interno mentre lo stato Loaded avrà una variabile contenente la lista dei todo.

First of all we start defining and naming the possible states of the list of todos. These state are only two: TodosLoadingState and TodoLoadedState. The Loading state indicates that the list of todos is still loading. The loaded state ,instead. Indicates that the list of todos has been successfully fetched from the database and is available. In order to define these two state a new abstract class is created first. It is called TodosState. It must extend the class Equatable. The class Equatable is useful to define equality between states without the need to override the equality operator in every state class. The TodosLoadingState does not contains other informations. The TodosLoadedState contains instead, a List variable filled with todos.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
abstract class TodosState extends Equatable{  
 const TodosState();  
  
 @override  
 List<Object> get props => [];  
}  
class TodosLoadingState extends TodosState{  
 @override  
 String toString() => 'TodosState - TodosLoadingState';  
}  
class TodosLoadedState extends TodosState{  
 final List<Todo> todos;  
 const TodosLoadedState(this.todos);  
  
 @override  
 List<Object> get props => [todos];  
  
 @override  
 String toString() => 'TodosState - TodosLoadedState';  
}   
\end{minted}

\mbox{}

\end{code}

**The state of the filtered list and the filter**

Also in this case there are only two possible states: FilteredTodosLoadingState and FilteredTodosLoadedState. The loading state identifies the fact that the filtered list hasn’t been computed (or todos fetched) yet. The Loaded state, instead, identifies the fact that the list of todos has been successfully fetched and the list of filtered todos computed. It contains two variables: a VisibilityFilter and a List of todos. An abstract class , called FilteredTodosState, must be created and extended with Equatable class. All other state classes ,of the filtered list’s state, will extend the FilteredTodosState class. Someone can notice that , the state of the filtered list and the filter ,contains two aspects of the application state: the filter and the filtered list. In this case it is possible to further split the state and create two separated blocs ,handling respectively the filter and the filtered list. From a general point of view ,the state should be divided in few possible pieces to keep things well separated and clean , like we do for classes and methods. However, the bloc pattern do not specify how granular should be the state fragmentation and , theoretically, we could decide to use a single bloc to handle the whole application ‘s state , like in Redux. In this particular case , I decided to implement a trade of and keep the filter and the filtered list in the same bloc. They concern two similar aspects of the data and ,splitting them , would require the bloc of the filtered todos to depend on the bloc of the filter, making its dependencies going from one bloc to two blocs ( the bloc of the todos and the bloc of the filter).

abstract class FilteredTodoState extends Equatable {  
 const FilteredTodoState();  
  
 @override  
 List<Object> get props => [];  
}  
  
class FilteredTodoLoadingState extends FilteredTodoState {  
 @override  
 String toString() => 'FilteredTodoState - FilteredTodoLoadingState';  
}  
  
class FilteredTodoLoadedState extends FilteredTodoState {  
 final List<Todo> todos;  
 final VisibilityFilter filter;  
  
 const FilteredTodoLoadedState(this.todos, this.filter);  
  
 @override  
 List<Object> get props => [todos, filter];  
  
 @override  
 String toString() => 'FilteredTodoState - FilteredTodoLoadedState';  
}

**The state of the stats**

Also in this case there only two possible states: StatsLoadingState and StatsLoadedState. The first identifies the fact that stats hasn’t been computed yet and do not contains any additional information inside. The second identifies the fact that stats are available and contains an int variable inside , called completed.

abstract class StatsState extends Equatable {  
 const StatsState();  
  
 @override  
 List<Object> get props => [];  
}  
  
class StatsLoadingState extends StatsState {  
 @override  
 String toString() {  
 return 'StatsState - StatsLoadingState';  
 }  
}  
  
class StatsLoadedState extends StatsState {  
 final int completed;  
  
 const StatsLoadedState(this.completed);  
  
 @override  
 List<Object> get props => [completed];  
  
 @override  
 String toString() {  
 return 'StatsState - StatsLoadedState : {completed: $completed}';  
 }  
}

**The state of the tab**

In order to define the states of the tab an enumeration is used.

enum TabState{  
 todos,stats  
}

**Events**

Now that states for every possible part of the application have been defined it’s the turn of Events. Events are just classes. They represent a specific actions the user can perform or also internal changes. They enable the states to mutate and create transitions.

**Events of the list of todos**

For the moment is sufficient to define two states only. One identifies the action of fetching todos from the database and is called LoadTodosEvent. It do not contains any other information. The other identifies the action of changing the completed field of a specific todo and is called SetCompletedTodoEvent. It contains two informations, the id of the specific todo and the new value for the completed field.

Also in this case a new abstract class is defined and extended with Equatable class. It is called TodosEvent. All other event classes concerning the state of the list of todo are extended with this class.

abstract class TodosEvent extends Equatable {  
 const TodosEvent();  
  
 @override  
 List<Object> get props => [];  
}

class LoadTodosEvent extends TodosEvent {  
 @override  
 String toString() => 'TodosEvent - LoadTodosEvent';  
}  
  
class SetCompletedTodoEvent extends TodosEvent {  
 final int id;  
 final bool completed;  
  
 const SetCompletedTodoEvent(this.id, this.completed);  
  
 @override  
 String toString() => 'TodosEvent - SetCompletedTodoEvent';  
}

**Events for the filtered list and the filter**

Two events are enough to define all possible transition for the state of the filtered list and the filter. One is called FilteredTodoChangeFilterEvent and is used to change the state of the filter. It contains , indeed, a VisibilityFilter variable that indicates the new value for the filter. The other event is called TodosUpdatedEvent . It informs the part of the state concerning the filtered list that the list of todo has changed and a new filtered list must be computed and a new FilteredTodosLoadedState emitted. It contains internally a variable providing the new list of todos.

Also in this case ,all event classes extend a common abstract class called FilteredTodoEvent which , in turn, extends the Equatable class.

abstract class FilteredTodoEvent extends Equatable {  
 const FilteredTodoEvent();  
  
 @override  
 List<Object> get props => [];  
}  
  
class FilteredTodoChangeFilterEvent extends FilteredTodoEvent {  
 final VisibilityFilter filter;  
  
 const FilteredTodoChangeFilterEvent(this.filter);  
  
 @override  
 List<Object> get props => [filter];  
  
 @override  
 String toString() => 'FilteredTodoEvent - FilteredTodoChangeFilterEvent {filter: $filter}';  
}  
  
class TodoUpdatedEvent extends FilteredTodoEvent {  
 final List<Todo> todos;  
  
 @override  
 List<Object> get props => [todos];  
  
 const TodoUpdatedEvent(this.todos);  
  
 @override  
 String toString() => 'FilteredTodoEvent - TodoUpdatedEvent';  
}29

**Events for the stats’s and tab’s state**

Both the state of the tab and the state of the stats require just one event. The event concerning the state of the tab is called ChangeTabEvent and contains internally a variable of type TabState indicating the value of the new tab. The event concerning the state of the stats is called StatsUpdatedEvent and is generated after the fetching or the updating of the list of todos. It contains internally the new list of todos.

Also in this case , both the event for the stats and the event for the tab extends respectively the abstact classes TabEvent and StatsEvent.

abstract class StatsEvent extends Equatable{  
 const StatsEvent();  
  
}  
class StatsUpdatedEvent extends StatsEvent{  
  
 final List<Todo> todos;  
 const StatsUpdatedEvent(this.todos);  
  
 @override  
 List<Object> get props => [todos];  
  
 @override  
 String toString() => 'StatsEvent - StatsUpdatedEvent';  
}

abstract class TabEvent extends Equatable{  
  
 const TabEvent();  
  
}  
  
class ChangeTabEvent extends TabEvent{  
 final TabState tab;  
  
 const ChangeTabEvent(this.tab);  
  
 @override  
 List<Object> get props => [tab];  
  
 @override  
 String toString() => 'TabUpdated { tab: $tab }';  
  
}

**The Blocs**

At this point both the events and the states ,necessary to implement di base functionalities of the application ,have been defined. Is possible ,then, to implement the classes, called blocs, that are going to define how new states are emitted in relation to the received events.

**The bloc for the list of todos**

To define the bloc for the list of todos is necessary to create a new class, that we name TodoBloc, and make it extends the Bloc class provided by the flutter\_bloc package. Moreover, it is necessary to provide , in the extension, also the type of events and states the bloc will manage. In our case , the ToboBloc class handles events of type TodosEvent and states of type TodosState, previously defined. A constructor must be defined inside which the bloc is initialized with a initial state. The initial state for the TodoBloc is of type TodoLoadingState by the fact that, at the application start, todos are still to be fetched from the database.

The Bloc class provided by the solution requires to override the “mapEventToState” method . The method is ,indeed, annoted with the @override notation meaning that the implementation we are giving substitutes the one in the class Bloc. The override is mandatory. The method mapEventToState takes as argument an event of type TodosEvent and returns a Stream of TodosStates. It is asynchronous ( indicated by the async\* annotation after the arguments) and do not terminate during the entire execution of the application. It keeps listening for new events, tough. Inside its implementation , a series of nested “if-else” structures have the task of identifing the type of the received event and to emit the consequent state. Indeed, the received event is always of the abstract type TodosEvent but can be of the subtype LoadTodosEvent or SetCompletedTodoEvent. Once the subtype is defined the event logic is processed and the new state emitted. The syntax yield\* Is used , Instead of the classic syntax “return”,because it allows to emit a new state, in the Stream , without terminating di mapEventToState method execution. If the return syntax is used , indeed, the new state is emitted correctly but the method is terminated and the application become unresponsive. For code readability, the logic to be executed when a to LoadTodoEvent or a SetCompletedTodoEvent is received has been moved to two other private methods called respectively mapLoadTodoToState and mapSetCompletedToState. This kind of practice is used also in the subsequence bloc implementation. The mapLoadTodoToState method takes as single argument an event of type LoadTodosEvent ( not a generic TodosEvent anymore) and bothers to fetch the todos from the database using the TodoRepository class. In case it successfully gets the list of todos it emits a new state of type LoadedTodoState containing the list of todos. In case of failure ,instead, the TodosLoadingState is emitted.

The mapSetCompletedToState takes as single argument anevent of type SetCompletedTodoEvent. After checking that the current state is of type TodosLoadedState ( in case it is not is meaningless to update the todo not having an actual list) a new list of todo is created containing the same todos as before except for the one with the id matching the one contained in the event. That todo, indeed, is replaced with a new one with the completed field set to the completed value contained in the event. Notice that a new instance of the list must be created and provided to the new state. If we just mutate the previous state’s list the Equatable class do not identify any difference between the previous state and the new emitted one, and so, do not notify all listeners.

class TodoBloc extends Bloc<TodosEvent, TodosState> {  
 TodoBloc() : super(TodosLoadingState());  
  
 @override  
 Stream<TodosState> mapEventToState(TodosEvent event) async\* {  
 if (event is LoadTodosEvent) {  
 yield\* \_mapLoadTodosToState(event);  
 } else if (event is SetCompletedTodoEvent) {  
 yield\* \_mapSetCompletedToState(event);  
 }   
 }  
  
 Stream<TodosState> \_mapLoadTodosToState(LoadTodosEvent event) async\* {  
 try {  
 final List<Todo> todos = await TodoRepository.*loadTodos*();  
 yield TodosLoadedState(todos);  
 } catch (e) {  
 yield TodoLoadingState();  
 }  
 }

Stream<TodosState> \_mapSetCompletedToState(  
 SetCompletedTodoEvent event) async\* {  
 if (state is TodosLoadedState) {  
 List<Todo> newList = (state as TodosLoadedState)  
 .todos  
 .map((todo) => todo.id == event.id  
 ? Todo(  
 name: todo.name,  
 description: todo.description,  
 id: todo.id,  
 completed: event.completed)  
 : todo)  
 .toList();  
 yield TodosLoadedState(newList);  
 }  
}

}

**The bloc for the filtered list and the filter**

The procedure is the same utilized for the todo bloc. A new class called FilteredTodosBloc is created and extended with the Bloc class. This new class handles the events of type FilteredTodosEvent and the states of type FilteredTodosState. Being the bloc of the filtered list of todos dependent from the bloc of the list of todos, an instance of this second one is passed inside the constructor and used later to listen for changes.

The instance of the bloc of the todos is saved in a local variable of type TodoBloc. In this case the constructor is a bit more articulated respect with before. It emits the initial state based on the state of the bloc of the todos. If it is a loaded state, the constructor computes and then emits a state of type FilteredtodosLoadedState using a filter of type all. If it is a loading state the constructor emits a state of type FilteredLoadingState.

class FilteredTodoBloc extends Bloc<FilteredTodoEvent, FilteredTodoState> {  
 final TodoBloc todoBloc;  
  
 FilteredTodoBloc({required this.todoBloc})  
 : super(  
 todoBloc.state is TodosLoadedState  
 ? FilteredTodoLoadedState(  
 (todoBloc.state as TodosLoadedState).todos,  
 VisibilityFilter.all,  
 )  
 : FilteredTodoLoadingState(),  
 )

}13

In addition, the constructor must register the bloc to the changes in the bloc of the list of todos. To do so the a particular variable of the TodoBloc instance is used. The variable is called “stream” and is present because the TodoBloc extends the Bloc class. It is , indeed, the stream variable where the mapEventToState method emits new states. We can register to its output using the method “listen”. Inside the listen method’s call a function must be provided. This function Is called everytime the steam value emits a new state. Inside this function the new emitted state can be accessed and used to implement some logic. Actually , we won’t implement the logic there but ,insteadm emit a new event that will be handled by the mapEventToState method defined later. In our case , the constructor registers to the TodoBloc stream . Once a new state is emitted it check if the state is of type TodoLoadedState. In case it is, it means that a new list of todos Is available. It can be the case that the list of todos has been just fetched or some todos have been update. In both of them,the bloc of the filtered list must compute a new filtered list and emit it through a new state. A specific event has been defined for this situation HERE RIFERIMENTo when the events related to the bloc of the filtered list has been implemented.

class FilteredTodoBloc extends Bloc<FilteredTodoEvent, FilteredTodoState> {  
 final TodoBloc todoBloc;  
 late StreamSubscription todoSubscription;  
  
 FilteredTodoBloc({required this.todoBloc})  
 : super(  
 todoBloc.state is TodosLoadedState  
 ? FilteredTodoLoadedState(  
 (todoBloc.state as TodosLoadedState).todos,  
 VisibilityFilter.all,  
 )  
 : FilteredTodoLoadingState(),  
 ) {  
 todoSubscription = todoBloc.stream.listen((state) {  
 if (state is TodosLoadedState) {  
 add(TodoUpdatedEvent((todoBloc.state as TodosLoadedState).todos));  
 }  
 });  
 }

It is the time to override the mapEventToState method defining the logic used to emit new state based on the received event. Like in the TodoBloc also In this case the method is asynchrounous and has as return type a stream of FilteredTodosState. The method takes as argument a single event of the generic abstraxt type FilteredTodosEvent. Inside the method, two nested “if-else” structures defines the type of the received event. There are only two possibility, indeed. The event can be of type FilteredTodosChangeFilterEvent or TodosUpdatedEvent. In the first case the private method mapTodoChangeFilterEventToState is called. In the second case the private method mapTodosUpdatedEventToState is called.

@override  
Stream<FilteredTodoState> mapEventToState(FilteredTodoEvent event) async\* {  
 if (event is FilteredTodoChangeFilterEvent) {  
 yield\* \_mapTodoChangeFilterEventToState(event);  
 } else if (event is TodoUpdatedEvent) {  
 yield\* \_mapTodoUpdatedEventToState(event);  
 }  
}8

The mapTodoChangeFilterEventToState method first checks that the state of the bloc of todos is of type TodosLoadedState ( in case it is not changing the filter is useless) and then it emits a new state of type FilteredTodosLoadedState containing the new filter and the new computed list of filtered todo using the new filter.

Stream<FilteredTodoState> \_mapTodoChangeFilterEventToState(  
 FilteredTodoChangeFilterEvent event) async\* {  
 if (todoBloc.state is TodosLoadedState) {  
 yield FilteredTodoLoadedState(  
 filterTodos((todoBloc.state as TodosLoadedState).todos, event.filter),  
 event.filter);  
 }  
}

The method mapTodoUpdatedEventToState checks that the TodoBloc’s state is of type TodosLoadedState and then emits a new state of type FilteredTodosLoadedState. The emitted state uses and contains the current filter ,if it is set, otherwise used the “all” filter.

Stream<FilteredTodoState> \_mapTodoUpdatedEventToState(  
 TodoUpdatedEvent event) async\* {  
 final filter = (state is FilteredTodoLoadedState)  
 ? (state as FilteredTodoLoadedState).filter  
 : VisibilityFilter.all;  
 if (todoBloc.state is TodosLoadedState) {  
 yield FilteredTodoLoadedState(  
 filterTodos((todoBloc.state as TodosLoadedState).todos, filter),  
 filter);  
 }  
}

The last thing to do is to ensure that the subscription to the todoBloc is disposed when the current bloc terminates.

@override  
Future<void> close() {  
 todoSubscription.cancel();  
 return super.close();  
}5

**The bloc for the stats**

This bloc is similar to the previous one except for the fact it has to deal with one event only: the StatsUpdatedEvent. Like usual the class StatsBloc is defined an extended with the Bloc class. The StatsBloc class handles event of the type StatEvent and states of the type StatsState. Also in this case the bloc depends on the bloc of the list of todo. For this reason a variable of type TodoBloc is added and initialized in the constructor. Also in the constructor a new initial state of type StatsLoadeingState is emitted. The subscription to the state stream of the TodoBloc is perfomed passing a function called onTodosStateChanged that check if the TodoBloc’s state is of type TodoLoadedState and , in case ,emits a event of type StatsUpdatedEvent. This event will be handled by the mapEventToState method implemented later. The function onTodosStateChanged is called also once in the constructor to update the stats in case the TodoBloc is already in a TodosLoadedState on StatsBloc creation.

class StatsBloc extends Bloc<StatsEvent, StatsState> {  
 final TodoBloc todoBloc;  
 late StreamSubscription todoSubscription;  
  
 StatsBloc({required this.todoBloc}) : super(StatsLoadingState()) {  
 void onTodosStateChanged(state) {  
 if (state is TodosLoadedState) {  
 add(StatsUpdatedEvent(state.todos));  
 }  
 }  
  
 onTodosStateChanged(todoBloc.state);  
  
 todoSubscription = todoBloc.stream.listen(onTodosStateChanged);  
 }

The mapEventToState method requires a single if statement because the only event is has to handle is the StatsUpdatedEvent. When received, the new stats are computed using the list of todo inside it and a new StatsLoadedState is emitted.

In the close method the subscription to the TodoBloc is terminated.

@override  
 Stream<StatsState> mapEventToState(StatsEvent event) async\* {  
 if (event is StatsUpdatedEvent) {  
 final numCompleted =  
 event.todos.where((todo) => todo.completed).toList().length;  
 yield StatsLoadedState(numCompleted);  
 }  
 }  
  
 @override  
 Future<void> close() {  
 todoSubscription.cancel();  
 return super.close();  
 }  
}

**The bloc for the tab**

The procedure is the same as before. This time the bloc is really simple. After creating the class TabBloc and extending it to the Bloc class, the states and events it must handle are specified. In the contructor the initial state is initialized and set to TabState.todos. The mapEventToState method is overridden connecting the only event with the emission of the state corresponding to the event TabState value.

class TabBloc extends Bloc<TabEvent,TabState>{  
 TabBloc() : super(TabState.todos);  
  
 @override  
 Stream<TabState> mapEventToState(TabEvent event)async\*{  
 if(event is ChangeTabEvent){  
 yield event.tab;  
 }  
 }  
  
}

**Observe blocs**

Terminates here the definition of the application’s state. All states ,events and blocs connecting them have been defined. It is possible to start testing the logic of the application, in the main function for example, initializing an object of type TodoBloc and trying to emit new events using the Add method offered by the Bloc package.

void main() {  
   
 TodoBloc todoBloc= TodoBloc();  
 todoBloc.add(LoadTodosEvent());  
   
}

The fact that it is possible to test the logic of the application without the need of writing a single widget explains how powerful is the bloc package. Is , indeed, really easy to split the logic layer from the presentation layer without dealing with complicated external dependencies. Moreover, it is possible to use an additional tool that helps the debugging and testing process; the BLocObserver. This component allows to intercept events, transitions and errors during the usage of the blocs and to execute arbitrary code when they occur. To use this component is necessary to define another class , that we call AppBlocObserver, and extend it with the BlocObserver class from the bloc package. Inside the AppBlocObserver class is possible to override three methods: “onEvent, onTransition and onError. onEvent is called everytime a new event is emitted in a bloc and provides, in its implementation ,the emitted event and the corresponding bloc. onTransition is called everytime a state transition inside a bloc occurs. Is offers two elements inside its implementation: the corresponding bloc and a object of type Transition. An object of type Transition is composed by two states and an event. The states are the one preceeding and postponing the the event execution. (note: not always the emission of a event produces a state transition. Some events may not generate a new state or may be ignored). Lastly, the method onError is called when an unexpected behaviour occurs and provides , in its implementation, the corresponding bloc where the error occeured and an object of type StackTrace that report the stack situation when the error occurred. In our case the corresponding event, transition and error are displayed only but other, more articolaed , implementation can be procided.

class AppBlocObserver extends BlocObserver{  
 @override  
 void onEvent(Bloc bloc, Object? event) {  
 super.onEvent(bloc, event);  
 print("Event : " +event.toString());  
 }  
  
 @override  
 void onTransition(Bloc bloc, Transition transition) {  
 super.onTransition(bloc, transition);  
 print( transition.toString());  
 }  
 @override  
 void onError(BlocBase bloc, Object error, StackTrace stackTrace) {  
 print(error);  
 super.onError(bloc, error, stackTrace);  
 }  
}

Before running the application with the runApp method ,the AppBlocObserver we just created is set as the default observer for the blocs.

void main() async {  
  
 Bloc.observer = AppBlocObserver();  
}

**Making the state accessible**

Similarly to the implementation with Redux and Inheritedwidget , also in this case, a particular widget called BlocProvider must be used to make the state , or part of it, accessible in the subtree. Since the information regarding the list of todos needs to be accessible by the entire application its BlocProvider is positioned in the root. In the main function , the first widget to be passed to the runApp method is indeed a BlocProvider. BlocProvider is a typed widget , meaning that the type of the bloc it is making accessible in the subtree must be provided. In our case it needs to provide a bloc of type TodoBloc. Inside the BlocProvider widget , two field must be filled: create and child. In the create field a function taking as single argument the context and returning a bloc of the specified type must be provided. This function is executed on the BlocProvider initialization. However, the initialization of the BlocProviders is lazy. This means that it is performed when the BlocProvider is accessed the first time and not when it is inserted in the tree. This type of procedure is used to postpone heavy methods execution as lately as possible to avoid, in case they are never accessed, to perform useless computation and waste time. We provide a function that instantiates a TodoBloc and emits the first event of the application: the LoadTodoEvent. In order to emit new events the method “add”, provided by the extension to Bloc class, is used. Moreover , the “cascade” notation offered by Dart language is used to increase de readability of the code. It allows to concatenate more actions/line of code using the pattern “..”.

The child field is populated with the MyApp widget as usual.

void main() async {  
  
 Bloc.observer = AppBlocObserver();  
 runApp( BlocProvider<TodoBloc>(create:(context)=> TodoBloc()..add(LoadTodosEvent()),child: const MyApp()));  
}3

Beyond the TodoBloc also the other blocs previously defined must be made accessible. They are required in the HomePage only because the information they provide are not used by the other pages. This time a MultiBlocProvider , instead of the BlocProvider, is used to wrap the HomePage. A MultiBlocProvider is nothing else that a widget itself that contains a field called “providers” where a list of BlocProvider widgets must be inserted. It is the same as nesting a series of BlocProviders but it makes the code more readable. In the list thre BlocProvider widgets are inserted. The first of type TabBloc, the second of type StatsBloc and the third of type FilteredTodoBloc. The last two must be initialized passing a TodoBloc in the constructor. In order to retrieve the TodoBloc the “of” method provided by the BlocProvider widget is used. The of method is called indicating the type of bloc to be searched and looks the bloc in the current context. It rises an error in case a bloc of the specified type is not found in the context. Fortunately , we already set a BlocProvider of type TodoBloc In the parent widget and so the of method successfully finds it in the current context. The reason because the TodoBloc is positioned in an higher level with respect to the other blocs it that ,it is a good practice to limitate the access to the state to the few parts of the application possible. This allow the state to be modified only by the parts that has access to it and , in case of problems, it is easier to understand which part of the code caused it.

class MyApp extends StatelessWidget {  
 const MyApp({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("building: MATERIAL-APP");  
 return MaterialApp(  
 initialRoute: "/",  
 routes: {  
 "/": (context) => MultiBlocProvider(providers: [  
 BlocProvider<TabBloc>(create: (context) => TabBloc()),  
 BlocProvider<StatsBloc>(  
 create: (context) =>  
 StatsBloc(todoBloc: BlocProvider.*of*<TodoBloc>(context))),  
 BlocProvider<FilteredTodoBloc>(  
 create: (context) =>  
 FilteredTodoBloc(todoBloc: BlocProvider.*of*<TodoBloc>(context))),  
 ], child: const HomePage()),  
 "/addTodo": (context) => const AddTodoPage(),  
 "/updateTodo" : (context) => UpdateTodoPage(todo: (ModalRoute.*of*(context)!.settings.arguments as Todo)),  
 },  
 );  
 }  
}

**State injection in the UI**

Now that the application state has been defined and also made accessible in the interested subtrees is the moment to connect it with the UI.

**The HomePage**

The Scaffold widget is wrapped into a BlocBuilder widget. The BlocBuilder widget is used to access the state concerning the tab. Indeed, Almost the entire HomePage is rebuilt when a tab change occurs. The entire HomePage creation is moved inside the “builder” field of the BlocBuilder widget. Moreover, the type of the bloc and the type of states the BlocBuilder has to manage are specified in the declaration. Inside the function of the “builder” field ,indeed, we have access to the state in the form of an object of the type previously provided, in addition to the current context.

class HomePage extends StatelessWidget {  
 const HomePage({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("building: HomePage");  
  
 return BlocBuilder<TabBloc, TabState>(builder: (context, tabState) {  
 });  
 }  
}

It is possible now to access the state of the tab through the variable tabState of type TabState and use it to build the Scaffold consequently.

builder: (context, tabState) {  
 return Scaffold(  
 appBar: AppBar(  
 title: const Text("TodoApp"),  
 actions: [tabState == TabState.todos? VisibilityFilterComponent():Container()],  
 ),  
 body: tabState == TabState.todos ? const TodoView() : const Stats(),  
 bottomNavigationBar: const TabSelector(),  
 floatingActionButton:  
 tabState == TabState.todos  
 ? FloatingActionButton(  
 child: const Icon(Icons.*plus\_one*),  
 onPressed: () {  
 Navigator.*pushNamed*(context, "/addTodo");  
 })  
 : Container()  
  
 );  
}

**The TodoView Component**

The TodoView component needs to access the state of the filtered list and the filter only. The ListView widget is wrapped in a BlocBuilder widget . We define by the <> notation that it will handle the bloc of type FilteredTodosBloc and its internal state (of type FilteredTodosState). In the function passed in the “builder” field the state is accessible using the variable called filteredTodosState. The actual type of the state is defined using a if statement. In case the state is of type FilteredTodosLoadingState a CircularProgressIndication widget is returned. In case the state is of type FilteredTodosLoadedState a variable containing the list of todos will be available in the filteredTodosState object and can be used to populated the ListView widget.

class TodoView extends StatelessWidget {  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return BlocBuilder<FilteredTodoBloc, FilteredTodoState>(  
 builder: (context, filteredTodoState) {  
 print("building: TodoView");  
  
 if (filteredTodoState is FilteredTodoLoadedState) {  
 return ListView.builder(  
 itemCount: filteredTodoState.todos.length,  
 itemBuilder: (context, index) {  
 return TodoItem(  
   
 todo: filteredTodoState.todos.elementAt(index));  
 });  
 } else if (filteredTodoState is FilteredTodoLoadingState) {  
 return const Center(child: CircularProgressIndicator());  
 } else {  
 return const Center(child: CircularProgressIndicator());  
 }  
 });  
 }  
}

11

**The TodoItem component**

Since this part of the development process do not consider any type of optimization, the TodoItem component does not need to be modified with respect to the implementation defined in RIFERIMENTO. The todo instance to be visualized is passed as argument in the constructor from the ancestor widget (TodoView). However, even if the TodoItem component does not access the state to read any value it needs to access the state to emit an event. Once the checkbox is tapped ,indeed, the list of todos should be modified. Emitting an event is easier than reading the state. It can be considered a constant action meaning that the widget should not be notified when the state changes. For this reason there is no need to use any BlocBuilder widget. It is sufficient to access the bloc in which the event must be emitted ,using the BlocProvider’s of method , and emit the event using the “add” method. The Checkbox widget’s onChanged function provides a Boolean variable (called completed in our case) that represents the value the Checkbox will take after being clicked. A new event of type SetCompletedTodoEvent is created using this variable and the id of the todo instance passed by the parent and emitted in the TodoBloc.

onChanged: (completed) {  
 BlocProvider.*of*<TodoBloc>(context)  
 .add(SetCompletedTodoEvent(id, completed!));  
}),

Summarizing; once the Checkbox is pressed, inside a TodoItem , a new event in the bloc of the list of todos is generated. This event causes a state transition in the TodoBloc passing from the current state to a new state where the corresponding todo has been modified. Then, the bloc of the filtered list and the bloc of the stats, listening for changes in the TodoBloc, react emitting a new internal event (respectively of type TodoUpdatedEvent and StatsUpdatedEvent) . This event causes a state transition of the questioned blocs to a new state where the filtered list and the stats are computed using the new TodoBloc’s state. As a conseguence of the change in the FilteredTodosBloc state the TodoView component is notified and rebuild showing the update.

**The VisibilityFilterSelector component**

The visiblityFilterSelector compoenent depends only by the bloc of the filtered list and the filter. It just need to visualize the current filter and to update the state with a new filter. The DropdownButton is then wrapped inside a BlocBuilder widget and the BlocBuilder informed with the <> notation that will handle the bloc of type FilteredTodosBloc and the states of type FilteredTodoState.

return BlocBuilder<FilteredTodoBloc, FilteredTodoState>(  
 builder: (context, filteredTodoState) {

Inside the BlocBuilder “builder” method a new variable of type Visibilityfilter is created and initialized based on the state of the FilteredTodosBloc. In case the state is of type loaded the variable is initialized with the current filter value. In case the state is of type loading the variable is initialized with the value all.

final VisibilityFilter filter= filteredTodoState is FilteredTodoLoadedState? filteredTodoState.filter: VisibilityFilter.all;

Then, the DropdownButton is populated with the created filter variable. Notice that the function provided in the “onChanged” field of every DropdownMenuItem widget uses its internal filter value to create and emit a new event in the FilteredTodoBloc ,when tapped ,of the type FilteredTodoChangeFilterEvent.

onChanged: (filter) {  
 BlocProvider.*of*<FilteredTodoBloc>(context).add(FilteredTodoChangeFilterEvent(filter!));  
},8

**The TabSelector component**

The entire component depends only by the state of the tab. It need to read and alo write the state. The BottomNavigatorBar widget is wrapped inside a BlocBuilder widget which is teached in handling the bloc of type TabBloc and the states of type TabState.

return BlocBuilder<TabBloc, TabState>(  
 builder: (context, currTab) {  
 return BottomNavigationBar(  
 currentIndex: TabState.values.indexOf(currTab),

TheBottomNavigationBar’s onTap field is populated making it emit a new event of the type ChangeTabEvent inside the TabBloc after the user clicks.

onTap: (index)=>BlocProvider.*of*<TabBloc>(context).add(ChangeTabEvent(TabState.values.elementAt(index))),

3

**The Stats component**

Also in this case the only dependency the Stats compenent has is with the part of the state concerning the stats. The component is, therefore, connected using a BlocBuilder and specifying inside its definition the bloc and the states it will handle to the interested part of the state. The bloc the BlocBuilder will handle is of type StatsBloc and the states are of type StatsState. Inside the function provided to the “builder” field the type of the current state is checked. In case the state is of type StatsLoadedState a widget of type Text is returned after populating it with the “completed” field inside the state object. In case the state is of type StatsLoadingState a CircularProgressIndicator widget is returned indicating that the stats still needs to be computed.

return BlocBuilder<StatsBloc, StatsState>(  
 builder: (context, statsState) {  
 return statsState is StatsLoadedState ?Center(  
 child: Text(  
 statsState.completed.toString()),  
 ) : Center(child: const CircularProgressIndicator());  
 },  
);

6

**Conclusions**

All the baase functionalities of the application has been implemented and work fine. In the overall the development process did not face big issues. The only hard part wes to create the two bloc of the filter and the stats. Being them dependend by an external bloc they required a little bit of attention the be handled correctly with respect to the rest of the application. Although the process was linear it required a llot of lines of code and boilerplate.

In the follow some summarizing data re reported:

Time spent: 10-12 hours

Lines of code written/updated: 367

Classes/widget created: 24

Created files: 12

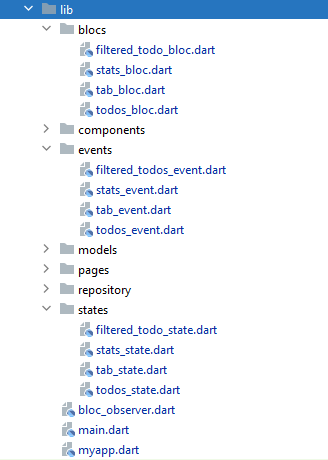
Tutte le funzionalità base dell’applicazione sono state sviluppate e funzionano correttamente. In generale il processo di sviluppo non ha avuto grossi intoppi. La l’unica parte complessa è stato creare i due blocchi del filtro e delle statistiche. Essendo essi dipendenti da un blocco esterno hanno richiesto un po’ più di attenzione per essere gestiti correttamente rispetto al resto dell’applicazione. Come si puo’ notare, sebbene lo sviluppo sia stato lineare, ha richiesto un numero notevole di linee di codice e di boilerplate. Riporto qui sotto alcuni dati riassuntivi:

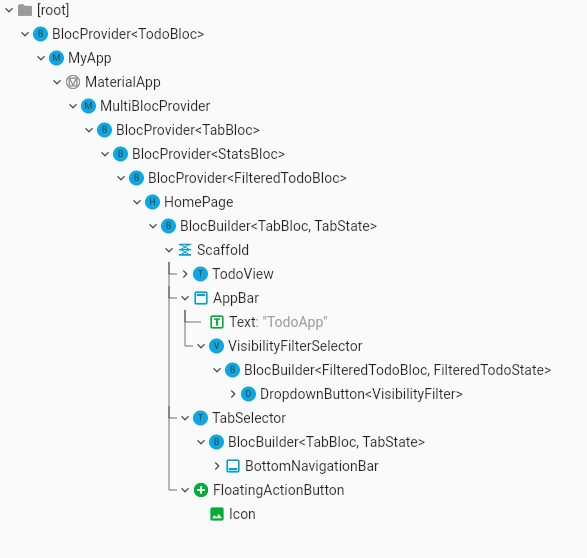
Tempo impiegato (per comprendere e scrivere la soluzione) : 10-12 ore

Linee di codice aggiunte/modificate: 367

Classi create: 24

File creati:12

 da inserire dopo la creazione del blocobserver



Da inserire nelle conclusioni

Immagine che contiene tavolo

Descrizione generata automaticamenteda inserire nelle ottimizzazioni

#### Features addition

New features presented HERE RIEFERIMENTO are added now to the base functionalities just implemented.

**\subparagraph**{ **New events**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The first thing to do is to make the state provide a way of adding and updating todos. Two new events are created and called AddTodoEvent and UpdateTodoEvent respectively. The AddTodoEvent will contain the name and the description to be used in the creation of the new todo instance. The id will be set by the method at the todo’s addition in order to generate a unique one. The completed field will be set to false by default being the new todo obviously pending at its origin. The UpdateTodoEvent will contain the id of the todo to be modified and the new name and description. The list of todos is contained in the TodoBloc . Moreover, the TodoBloc handles events of the type TodosEvent. Both the AddTodoEvent and the UpdateTodoEvent are extended ,tough, with the TodosEvent abstract class in order to be manageable by the TodoBloc.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class AddTodoEvent extends TodosEvent {  
 final String name;  
 final String desc;  
  
 const AddTodoEvent(this.name, this.desc);  
  
 @override  
 String toString() => 'TodosEvent - AddTodoEvent';  
}

class UpdateTodoEvent extends TodosEvent {  
 final int id;  
 final String newName;  
 final String newDesc;  
  
 const UpdateTodoEvent(this.id, this.newName, this.newDesc);  
  
 @override  
 List<Object> get props => [id, newName, newDesc];  
  
 @override  
 String toString() => 'TodosEvent - UpdateTodoEvent';  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **Bloc update**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

It necessary now to “teach” the TodoBloc at handling these new events. The workflow will be the following: when the AddTodoEvent is received in the TodoBloc , a new instance of the list of todos , contained in the current state , is generated . A new todo is created using the name and description contained in the event. The new todo is added to new instance of the list. The new list is then used to creted a new state of the type TodosLoadedState before emitting it in the TodoBloc. When the UpdateTodoEvent is received in the TodoBloc a new instance of the list contained in the state is created. The todo with the id matching the one contained in the event is modified using the new name and description. Lastly a state of type TodosLoadedState is emitted with thew new list. There is one more thing to do, adding to the TodoBloc’s mapEventToState method two new if-else branches that check if the received event is of type AddTodoEvent of UpdateTodoEvent. In the first case the private method mapTodoAddedToState is called passing the event as parameter. In the second case the private method mapTodoUpdatedToState is called, instead passing the event as parameter.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

@override  
Stream<TodosState> mapEventToState(TodosEvent event) async\* {  
 if (event is LoadTodosEvent) {  
 yield\* \_mapLoadTodosToState(event);  
 } else if (event is AddTodoEvent) {  
 yield\* \_mapTodoAddedToState(event);  
 } else if (event is UpdateTodoEvent) {  
 yield\* \_mapTodoUpdatedToState(event);  
 } else if (event is SetCompletedTodoEvent) {  
 yield\* \_mapSetCompletedToState(event);  
 }   
}

\end{minted}

\mbox{}

\end{code}

The mapTodoAddedToState method checks if the current state is of type TodosLoadedState ( if it is not is meaningless to perform any addition), then , creates a unique id and a new instance of the list of todos contained in the state variable. It then adds the todo to the new list, Lastly, the TodosLoadedState created with the new list is emitted.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

Stream<TodosState> \_mapTodoAddedToState(AddTodoEvent event) async\* {  
 if (state is TodosLoadedState) {  
 Random rand = Random();  
 List<int> ids =  
 (state as TodosLoadedState).todos.map((todo) => todo.id).toList();  
 int newId = rand.nextInt(1000) + 2;  
 while (ids.contains(newId)) {  
 newId = rand.nextInt(1000) + 2;  
 }  
 Todo newTodo = Todo(  
 id: newId,  
 name: event.name,  
 description: event.desc + " " + newId.toString(),  
 completed: false);  
 final List<Todo> updatedTodos =  
 List.from((state as TodosLoadedState).todos)..add(newTodo);  
 yield TodosLoadedState(updatedTodos);  
 }  
}

\end{minted}

\mbox{}

\end{code}

The mapTodoUpdatedToState method checks if the current state is of type TodosLoadedState ( if it is not is meaningless to perform any modification), then , creates a new instance of the list of todos contained in the state variable and modify the todo matching the id. Lastly, the TodosLoadedState created with the new list is emitted.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

Stream<TodosState> \_mapTodoUpdatedToState(UpdateTodoEvent event) async\* {  
 if (state is TodosLoadedState) {  
 List<Todo> newTodos = (state as TodosLoadedState)  
 .todos  
 .map((todo) => todo.id == event.id  
 ? Todo(  
 id: event.id,  
 name: event.newName,  
 description: event.newDesc,  
 completed: false)  
 : todo)  
 .toList();  
 yield TodosLoadedState(newTodos);  
 }  
}

\end{minted}

\mbox{}

\end{code}

**\subparagraph**{ **Access new feature in the UI**}\mbox{}\\

**\label{subpar:todo\_app\_bloc\_core\_state}**

The state can now handle this new functionalities correctly. Thanks to the fact that we situated the TodoBloc on top of the widgets tree it is possible now to access the state in the AddTodoPage and in the UpdateTodoPage easily. An instance of the TodoBloc ,indeed , exists in the current context and can be accessed using the of method. In case the TodoBloc was located in a lower level with respect to the MaterialApp widget ( from where routes are create) it would have had to be passed by argument to the corresponding pages. However, being the TodoBloc accessible , it can be used in the AddTodoPage to perform the todo addition , at the TextButton’s push, using the parameters contained in the TextField widgets. The AddTodoPage implementation is reported RIFERIMENTO. The only part to be changed is the onPressed field of the TextButton widget. Inside the provided function, the TodoBloc instance is accessed using the of method and the AddTodoEvent emitted. The AddTodoPage is popped then to come back in the HomePage ,where the TodoView rebuilds due to the change of the state of the TodoBloc.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

TextButton(  
 onPressed: () {  
 BlocProvider.*of*<TodoBloc>(context).add(AddTodoEvent(  
 textControllerName.text, textControllerDesc.text));  
 Navigator.*pop*(context);  
 },  
 child: const Text("Create"))

\end{minted}

\mbox{}

\end{code}

The same process is done to implement the UpdateTodoPage. In the onPressed field a function is provided. This function uses the TextField widgets’s parameters to create and emit an event of type UpdateTodoEvent. The UpdateTodoPage is then popped and the HomePage rebuilt due to the TodoBloc state change.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

TextButton(  
 onPressed: () {  
 BlocProvider.*of*<TodoBloc>(context).add(UpdateTodoEvent(  
 widget.todo.id,  
 textControllerName.text,  
 textControllerDesc.text));  
 Navigator.*pop*(context);  
 },  
 child: const Text("Confirm"))

\end{minted}

\mbox{}

\end{code}

Conclusions

The features addition process was easy and did not required many lines of code.

**Time spent: 15-20 minutes**

**Lines of code written/updated: 67**

**Classes/widget created: 2**

#### Rendering optimizations

To achieve the desired partial rendering is necessary to work on the TodoView and TodoItem widgets only. We will leverage on a specific field of the BlocBuilder widget called buildWhen. In this field is possible to insert a Boolean function in order to determine whether or not to rebuild the questioned widget on a state transition. Inside this function , the previous state and the next state can be accessed. For the moment, when a state change occurs, the TodoView widget destroys all the children TodoItem widgets and rebuilds them using the data contained in the new state. Well, actually is not precisely like this. Flutter indeed uses some articulated mechanism in order to rebuild the few parts of the widget tree possible. From our prespective ,however, TodoItem widgets are destroyed and recreated at every transaction, also in case a single TodoItem changed. To make the TodoItem widgets self-rebuild , is necessary to make them sensible to changes in the state. For the moment , however, the TodoItem widget does not use any BlocBuilder widget at all. Indeed, it just visualize the todo passed from the parent widget without actually accessing the state. The first thing to do is to wrap the TodoItem widget inside a BlocBuilder widget making it listen for changes in the FilteredTodoBloc. Instead of receiving ,from the TodoView parent widget ,a copy of the todo instance to be visualized is enough ,now, to receive the id and use it to obtain the instance of the todo accessing the state. Inside the builder field, the function will look up for the corresponding todo in the list of filtered todos and use it to create the TodoItem.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class TodoItem extends StatelessWidget {  
 final int id; //Todo variable replaced with int  
  
 const TodoItem({Key? key, required this.id}) : super(key: key);

@override  
Widget build(BuildContext context) {

//added the BlocBuilder widget  
 return BlocBuilder<FilteredTodoBloc, FilteredTodoState>(  
builder: (context, state) {  
 print("building: Todo Item $id " + key.toString());  
  
 if (state is FilteredTodoLoadedState) {  
 Todo t = (state).todos.firstWhere((element) => element.id == id);  
 return InkWell(. . .);  
 } else {  
 return Row(  
 children: const [  
 Text("ERROR"),  
 ],  
 );  
 }  
});

\end{minted}

\mbox{}

\end{code}

At this point, both TodoVIew and TodoItem widgets listen for changes in the state. The buildWhen field ,introduced above , is used now to teach them when to rebuild. Starting from the TodoView widget we provide a function to the buildWhen field that checks the types of the previous and the next state. If they are different there is no need to proceed further and a true value is returned meaning that the TodoView widget must be rebuilt. If they are equal the function checks if the length of the previous filtered list and the length of the new one are the same. If they differs is necessary to rebuild the entire TodoView widget. In case they are equal the function proceeds checking if the elements contained in the first list are exactly the ones contained in the second. ( note: it checks the ids only because we are not interested in knowing if the other field changed, the TodoItem will check it later) If all the elements are the same there is no need to rebuild the TodoView and the function terminates returning false.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

return BlocBuilder<FilteredTodoBloc, FilteredTodoState>(  
 buildWhen: (previous, next) {  
 return !((previous is FilteredTodoLoadedState) &&  
 (next is FilteredTodoLoadedState) &&  
 previous.todos.length == next.todos.length &&  
 listEquals(next.todos.map((todo) => todo.id).toList(),  
 previous.todos.map((todo) => todo.id).toList()));  
},

\end{minted}

\mbox{}

\end{code}

The same must be done in the TodoItem widget. The TodoItem needs to be rebuilt when the new list still contains the todo matching its id and ,that todo ,changed one or more of its internal fields: the name , the description or the completed field. A function is provided in the buildWhen field that checks if the previous and the next states are both of type FilteredTodoLoadedState. In case they aren’t is useless to rebuild the TodoItem widget and so the false value is returned. In case they are the function checks if the new state contains the corresponding todo. In case it does not contains the todo it is useless to rebuild the TodoItem and so the false value is returned. In case it contains the todo the function checks if the todo changed its internal fields with respect to the previous state. In case it does the function returns true and the TodoItem is rebuilt.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - Bloc - states definition for the filtered list of todos and the filter} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

buildWhen: (previous, next) {  
 if (next is FilteredTodoLoadedState &&  
 previous is FilteredTodoLoadedState) {  
 if (next.todos.map((todo) => todo.id).toList().contains(id) == true) {  
 if (previous.todos.firstWhere((element) => element.id == id) ==  
 next.todos.firstWhere((element) => element.id == id)) {  
 return false;  
 }  
 } else {  
 return false;  
 }  
 }  
 return true;  
}

\end{minted}

\mbox{}

\end{code}

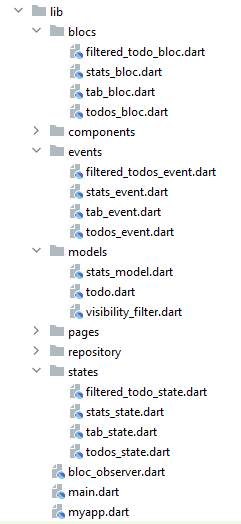
**Conclusions**

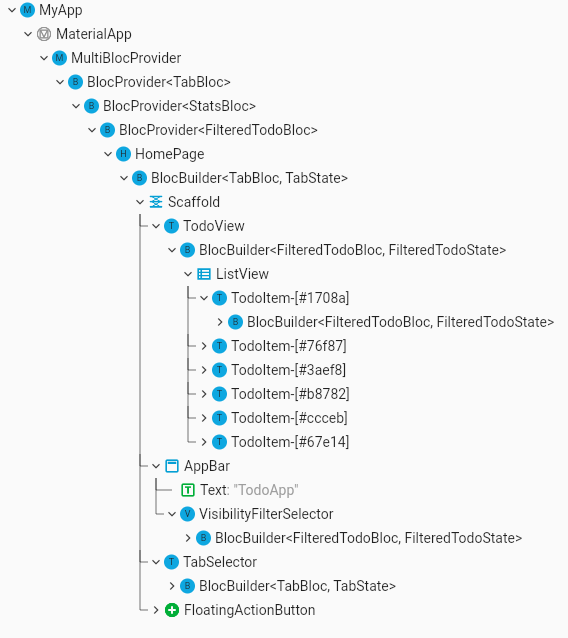
It was not an easy task to perform these optimizations.

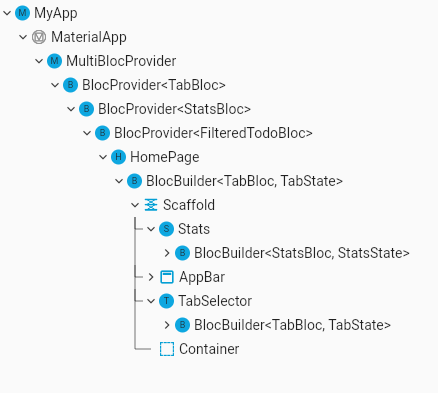
**Time spent: 6-8 hours**

**Lines of code written/updated: 33**

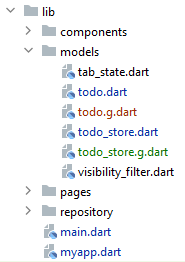
**Classes/widget created: 0**

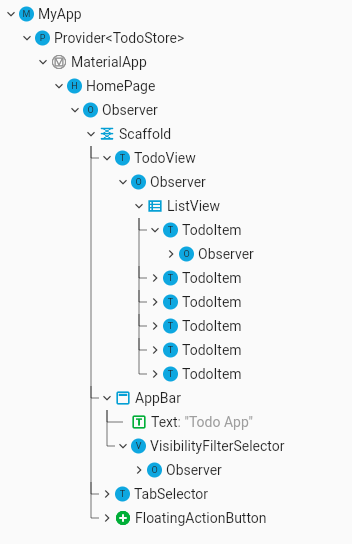


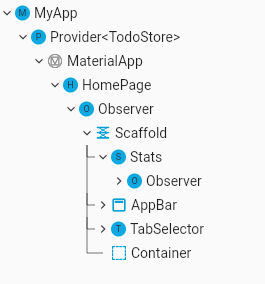




### MobX implementation

**\subsubsection{Base funtionalities}** \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**





As mentioned in the Chapter XX RIFERIMENTO the mobx package makes part of the state Observable and uses a particular widget called Observer to keep track of the changes of the observable variables. In order to be able to define observable parts is necessary to extend the class containing them with the Store class offedered by the package. Moreover the container class must be made abstract. The usual definition of the Todo class is used for the moment remembering that no rendering optimization is kept into account. The parts of the state to be made observable are the list of todos , the filter and the tab value. The whole application indeed relies on them to track changes and update correspondingly.

**\paragraph**{The observable state - }

**\label{subpar:todo\_app\_bloc\_core\_state}**A new abstract class is created and called \_TodoStore. It contains the list of todos and the filter. A separate observable variable will be used to implement the state of the tab. This design choice allows to show two different approaches the mobx package provides and divides the information regarding the todos from the information regarding the tab value. ( the filter value is in some way connected with the list of todo). This new abstract class is extended with the Store class. A list of todos and a visibility filter are created inside it and annoted with the @observable annotation. The annotation informs the code generator that those variables need to be observable and the code generator automatically creates a getter and a setter action for them.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
abstract class \_TodoStore with Store {  
 @observable  
 List<Todo> todos = [];  
  
 @observable  
 VisibilityFilter filter = VisibilityFilter.all;  
}

\end{minted}

\mbox{}

\end{code}

**\paragraph**{Actions - }

**\label{subpar:todo\_app\_bloc\_core\_state}**In this implementation the strict mode mobx provides is set to “always”. This choice reflects the common usage of the pattern. It is indeed a common choice to allow the state mutation only through actions. This behavior comes with a lot of advantages that make it the correct choice for the 99% of the cases. This subject will be deeper investigated in the latter sections but, for the moment , it is worth noting that the mobx package also provides the possibility to configure the strict mode to “never” not allowing to directly change the state without passing through an action. A bunch of new methods are added to the TodoStore class and marked with the @action annotation. The simplest one is the changeFilter method that allows to change the current filter.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

@action  
 void changeFilter(VisibilityFilter filter) {  
 this.filter = filter;  
 }

\end{minted}

\mbox{}

\end{code}

A method to set the completed field of a particular todo is also required. This method is called setCompleted and takes the id of the todo to be changed and its new completed value. As usual , a new list is created after modifying the todo in order to allow mobx to recognize the change in the list of todos.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

@action  
 void setCompleted(int id, bool completed) {  
 assert(todoExists(id) != null, 'No todo with id : $id');  
 todos = todos.map((element) {  
 if (element.id == id) {  
 return Todo(  
 completed: completed,  
 description: element.description,  
 name: element.name,  
 id: element.id);  
 } else {  
 return element;  
 }  
 }).toList();  
 }  
  
  
Todo? todoExists(int id) {  
 List<Todo> result = todos.where((element) => element.id == id).toList();  
 return result.isNotEmpty ? result.first : null;  
}

\end{minted}

\mbox{}

\end{code}

The two usual methods to fetch and save the todos into the DataBase/repository are implemented and annotated with the @action annotation.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
@action  
Future<void> fetchTodos() async {  
 todos = await TodoRepository.*loadTodos*();  
}  
  
@action  
Future<void> saveTodos() async {  
 await TodoRepository.*saveTodos*(todos);  
}

\end{minted}

\mbox{}

\end{code}

**\paragraph**{Computed fields - }

**\label{subpar:todo\_app\_bloc\_core\_state}**computed fields are part of the state derived from other parts of the state. They are a pivotal point in the mobx state management because the package is able to smartly compute them and performs lot of optimizations under the hood and uses memoization technique to prevent useless computations. They are a similar concept with respect to selector in Redux. The list of completed todos is well suited to demonstrate the power of the computed field. A new getter function is created and called completedTodos. It computes the list of completed todos and returns it. The @computed annotation is positioned right above the method to make the code generator know how to implement it in order to perform the optimizations discussed earlier. During the application lifecycle the completedTodos method will be accessed numerous times and automatically recomputed in case a part of the state ,it depends on, changes. Moreover, its value is memoized and reused in case multiple accesses are necessary. Another method is created in the same way to compute the pendingTodos.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
@computed  
List<Todo> get completedTodos =>  
 todos.where((element) => element.completed).toList();  
  
@computed  
List<Todo> get pendingTodos =>  
 todos.where((element) => !element.completed).toList();

\end{minted}

\mbox{}

\end{code}

pendingTodos and completedTodos methods are then used to implement another computed values, the filteredTodos, the number of completed todos and the number of pending todos. The filteredTodos method returns the list of todos that match the visibility filter. It is composed using the computed values defined before. Also the stats value can be obtained using the computed feature.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
@computed  
List<Todo> get filteredTodos {  
 switch (filter) {  
 case VisibilityFilter.all:  
 return todos;  
 case VisibilityFilter.completed:  
 return todos.where((element) => element.completed).toList();  
 case VisibilityFilter.notCompleted:  
 return todos.where((element) => !element.completed).toList();  
 }  
}

@computed  
int get len => todos.length;  
  
@computed  
int get completed => completedTodos.length;  
  
@computed  
int get pending => pendingTodos.length;  
  
@computed  
String get stats {  
 return completed.toString();  
}

\end{minted}

\mbox{}

\end{code}

**\paragraph**{The code generation - }

**\label{subpar:todo\_app\_bloc\_core\_state}**The \_TodoStore class won’t be directly used in the code, its actual implementation instead will. The \_TodoStore is indeed an abstract class and is used by the code generator to generate the actual TodoStore class. A particular line of code must be placed just below the imports to allow the code generator to . The generated code will be inside the part file: counter.g.dart, which isd included with the *part directive right below the imports*. Without this, the code generator will not produce any output. The generated file contains the \_$TodoStore mixin used with the \_TodoStore abstract class to finally implement the TodoStore.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

part 'todo\_store.g.dart';  
  
class TodoStore = \_TodoStore with \_$TodoStore;

\end{minted}

\mbox{}

\end{code}

In order to generate the code, a series of directives can be used into the terminal. For the sake of simplicity, the following line of code is always used.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

flutter pub run build\_runner watch --delete-conflicting-outputs

\end{minted}

\mbox{}

\end{code}

It automatically generate the code and handles all the possible conflicts that can arise. For example, in case a file named todo\_store.g.dart already exists it first deletes the file and then computes it again. With this line of code, the code generation process is made really easy. The drawback is that every time the generated code must be modified it is generated from scratch instead. In our case is not a big deal because the application is contained and the code generator only takes about 12-13 seconds to execute and generate the entire code. In a more spread scenario other directives can be used to make the code generation process lighter. I decided not to show the generated code because it is quite long and hard to read.

**\paragraph**{The spy feature - }

**\label{subpar:todo\_app\_bloc\_core\_state}** in order to enable the spy feature a new configuration for the mainContext must be provided before running the application in the main function. After cloning the default mainContext the isSpyEnabled field is set to true and the writePolicy is set to always in order to enable the strict-mode for every state change. Moreover, a function is passed to the spy feature where is possible to perform arbitrary actions when events occurs. In our case is enough to print the event name when an event of type action occurs to have a clearer picture of what is going on.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

mainContext.config = mainContext.config  
 .clone(isSpyEnabled: true, writePolicy: ReactiveWritePolicy.always);  
mainContext.spy((event) {  
 if (event.type == "action") {  
 print("event name : " + event.name);  
 }  
});

\end{minted}

\mbox{}

\end{code}

And that’s basically all we need in order to implement the application’s state. At this point is indeed already possible to test the state logic.

**\paragraph**{Making the state accessible - }

**\label{subpar:todo\_app\_bloc\_core\_state}** MobX ,like every other solution used so far ,uses a provider widget to supply the state to the subtree. In this case the mobx package do not self-implement the provider widget, instead it relies on and external package called Provider which offers this feature. The procedure is the usual one, the MaterialApp widget is wrapped into a Provider widget of type TodoStore which supplies the instance of the TodoStore to the subtree. It has a create field where the TodoStore can be initialized and where the fetching of the todos can take place.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

return Provider<TodoStore>(  
 create: (\_) => TodoStore()..fetchTodos(),  
 child: MaterialApp(. . .),  
 }));

\end{minted}

\mbox{}

\end{code}

**\paragraph**{The HomePage and the TabSelector component - }

**\label{subpar:todo\_app\_bloc\_core\_state}**The HomePage is almost entirely build using the state part regarding the tab. The first thing to do is to create an observable variable of type TabState to represent it. This time a different approach with respect to the one used to implement the state of the todos and the filter is used. We are not creating an abstract class neither generating any code. Moreover, no Provider widget is used because the tab value needs to be accessed only in a couple of widgets and passing it between them is way easier. The tab value is wrapped into an observable object of type TabState. The entire object is managed by the mobx package and the only difference with respect to a normal variable is that in order to access the TabState value we need to further dig into the value field instead of just using the tab variable as it is. Both the body and the visibility filter component as well as the TabSelector depends on the tab value so the entire Scaffold widget is wrapped into a Observer widget. Once a change in the tab variable occurs the Observer widget automatically determine which parts of the scaffold widget to rebuild. In our case all components depends on the tab value and then every component is rebuild once a tab change occurs, but in case some component were independent by the tab value and wrapped into the observer widget it would not be rebuilt. The entire Scaffold is populated using the tab variable as usual, the only part that differs is the TabSelector component. Beside depending on the tab value it also need to change it. There are two equivalent options : passing the tab variable to the TabSelector component or passing a closure function that changed the tab variable. The first solution is used. A new variable is added to the TabSelector widget and populated in the constructor. It is used in the onTap field’s function to mutate its value once the user taps on one specific BottomNavigationBarItem. Notice that the state change is wrapped into a runInAction object. This because if we just change the tab value directly, a run time error would arise alerting that observable values cannot be changed outside actions. This is due to the strict mode previously set. runInAction creates a throwaway action we can use directly in the code. This approach is made necessary because we used a stand alone Observable variable. If we included the tab variable into the TodoStore class the code generator would had automatically create also its getter and setter actions.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
class HomePage extends StatelessWidget {  
 HomePage({Key? key}) : super(key: key);  
 final \_tab = Observable<TabState>(TabState.todos);  
  
 @override  
 Widget build(BuildContext context) {  
 print("building HomePage");  
 return Observer(  
 builder: (context) {  
 return Scaffold(  
 appBar: AppBar(. . .),  
 body: \_tab.value == TabState.todos ? const TodoView() : const Stats(),  
 bottomNavigationBar: TabSelector(  
 tab: \_tab,  
 ),  
 floatingActionButton: \_tab.value == TabState.todos  
 ? FloatingActionButton(. . .)  
 : Container()  
  
 );  
 }  
 );  
 }  
}

\end{minted}

\mbox{}

\end{code}

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

class TabSelector extends StatelessWidget {  
 final Observable<TabState> tab;  
  
 const TabSelector({Key? key, required this.tab}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("building TabSelector");  
  
 return BottomNavigationBar(  
 currentIndex: TabState.values.indexOf(tab.value),  
 onTap: (index) {  
 runInAction(() => tab.value = TabState.values.elementAt(index));  
 },  
 items: TabState.values  
 .map((tab) => BottomNavigationBarItem(...))  
 .toList(),  
 );  
  
 }  
}

\end{minted}

\mbox{}

\end{code}

**\paragraph**{The VisibilityFilterSelector component - }

**\label{subpar:todo\_app\_bloc\_core\_state}**It entirely depends on the value of the filter in the TodoStore. To obtain the instance of the TodoStore we use the static of method of the Provider widget specifying the type of the instance we are looking for. This procedure will be frequently used from now on and is usually performed at the beginning of the build method.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

final store = Provider.*of*<TodoStore>(context);

\end{minted}

\mbox{}

\end{code}

The entire VisibilityFilterSelector component is then populated using the filter variable contained in the TodoStore as usual and wrapped into an Observer widget to make is responsive to filter changes. In the onChanged field of the DropdownMenuItem widgets the action changeFilter previously defined is used to

Set the filter value to the tapped one.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

return Observer(  
 builder: (context) {  
 print("building Visibilityfilter");  
  
 return DropdownButton<VisibilityFilter>(  
 value: store.filter,  
 items: VisibilityFilter.values.map((filter) {  
 return DropdownMenuItem<VisibilityFilter>(…);}).toList(),  
 onChanged: (tappedValue) {  
 store.changeFilter(tappedValue!);  
 },  
 );  
 },  
);

\end{minted}

\mbox{}

\end{code}

Notice that we could omit the usage of the changeFilter action and set the value of the filter directly. This because a setter and a getter action are automatically created by the code generator for every observable field. This implies that the changeFilter action could also be omitted in the definition of the TodoStore abstract class.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

onChanged: (tappedValue) {  
 store.filter=tappedValue!;   
 },

\end{minted}

\mbox{}

\end{code}

Personally , I dislike this feature mobx offers in the Flutter framework. I investigated a bit in the usage of MobX with React and JS finding that in that case it behaves as expected raising a warning in case field are directly changed. (violating the strict mode) I find the way of changing the filter value proposed in the Source Code RIFERIMENTO a lot more elegant with respect to the one proposed in the Source Code RIFERIMENTO. Explicitly using predefined actions brings way more meaning for the reader and prevents programmers to accidentally mutate the state in the implementation process. At the end , both approaches pass though actions to change the state and this respect the MobX approach. Actions are indeed really important to the correct functioning of the application because using them the mobx package can generate atomic state transitions. Suppose a simple action, like the one we just used , produces reactions of different types and those reactions affects different parts of the state and the UI. Suppose now that the strict mode is disabled and set to never. In that case can happen that the various reactions are completed/fired in different interval of time because their carried computation is heavier of lighter depending with respect of the other. Consequently the entire state of the application is not well synchronized and the UI could reflect this inconsistency bringing to some bad situations. Mobx package instead ensure that actions and consequent reactions are performed atomically without leaking any intermediate values as long as actions are used.

**\paragraph**{The Stats component - }

**\label{subpar:todo\_app\_bloc\_core\_state}** this component is really simple. It just needs to access the TodoStore to get the stats value. The procedure shown in the Source Code RIFERIMENTo is used as usual to get an instance of the TodoStore and consequently of the stats value. The entire widget is then wrapped into an Observer widget .

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

final store = Provider.*of*<TodoStore>(context);  
return Observer(  
 builder: (context) {  
 return Center(child: Text(store.stats));  
 },  
);

\end{minted}

\mbox{}

\end{code}

**\paragraph**{The TodoView component and the TodoItem component - }

**\label{subpar:todo\_app\_bloc\_core\_state}** These two components represent the core of the UI development process and no optimizations are considered for the moment. The TodoView widget needs to access the state and rerender once a change occurs in the filteredList of todos. In this scenario the mobx pattern really shines. It is sufficient to wrap the entire widget inside a Observer widget and access the filtered list of todo contained in the TodoStore to allow the mobx package to automatically detect a change in the filtered list. Once the list of todos or the filter is updated the filteredList is automatically computed. As usual the TodoItem just acts as a visualizer for the corresponding todo and so takes as argument the todo instate to be visualized.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

final store = Provider.*of*<TodoStore>(context);  
return Observer(  
 builder: (\_) {  
 print("building TodoView");  
  
 return ListView.builder(  
 itemCount: store.filteredTodos.length,  
 itemBuilder: (context, index) {  
 return TodoItem(  
 todo: store.filteredTodos.elementAt(index),  
 );  
 },  
 );  
 },  
);

\end{minted}

\mbox{}

\end{code}

The TodoItem widget implementation is basically the same presented HERE RIFERIMENTo except for the onChanged field of the TextButton widget that needs to be filled up. Inside the provided function the store is retrieved using the Provider widget and used to fire the setCompleted action using the todo’s id and the new value for the completed field. Notice ,to be precise, that also in this case the completed field could be changed directly without any error.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

onChanged: (value) {  
 final store = Provider.*of*<TodoStore>(context);  
 store.setCompleted(todo.id, value!);  
 *// todo.completed = value!;*}),

\end{minted}

\mbox{}

\end{code}

The base functionalities are now working fine.

**\subsubsection{Features addition}** \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**In order to add and update todos its necessary to define two new actions. Also in this case the creation of these two new actions could be avoided because setter and getter methods already exists for the list of todos. However , actions carry a lot more meaning and enable to avoid code duplication. Two new methods are added to the TodoStore class and annoted with the @action syntax. The functioning is the usual one, in the addTodo method a new todo instance is created with an unique id and added to the list of todos. In the updateTodo method the id argument is used to search for the corresponding todo and the name and desc argument are used to update it. Both methods need to substitute the todo’s list with a new instance to allow the mobx package to recognize the change. Subsequently the code is generated again using the line of code presented HERE RIFERIMENTO.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

@action  
void updateTodo(int id, String name, String desc) {  
 assert(todoExists(id) != null, 'No todo with id : $id');  
 Todo todo=todos.where((element) => element.id==id).first;  
 todo.name=name;  
 todo.description=desc;  
}

@action  
void addTodo(String name, String desc) {  
 Random rand = Random();  
 List<int> ids = todos.map((e) => e.id).toList();  
 int newId = rand.nextInt(1000) + 2;  
 while (ids.contains(newId)) {  
 newId = rand.nextInt(1000) + 2;  
 }  
 Todo newTodo = Todo(  
 id: newId,  
 name: name,  
 description: desc + " " + newId.toString(),  
 completed: false);  
  
 todos.add(newTodo);  
 todos = todos.toList();  
}

\end{minted}

\mbox{}

\end{code}

Accessing these new actions in the UpdateTodoPage and in the AddTodoPage is simple by the fact that the Provider widget has been situated as parent of the MaterialApp widget ( from where different roots generate). In the onPressed field of the TextButton situated in the AddTodoPage the store is retrieved using the Provider’s of method and used to fire a action of type addTodo.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

onPressed: () {  
 final store =  
 Provider.*of*<TodoStore>(context, listen: false);  
 store.addTodo(  
 textControllerName.text, textControllerDesc.text);  
 Navigator.*pop*(context);  
}

\end{minted}

\mbox{}

\end{code}

The same is done for the onPressed field of the TextButton in the UpdateTodoPage.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

onPressed: () {  
 final store =  
 Provider.*of*<TodoStore>(context, listen: false);  
 store.updateTodo(widget.todo.id, textControllerName.text,  
 textControllerDesc.text);  
 Navigator.*pop*(context);  
},

\end{minted}

\mbox{}

\end{code}

**\subsubsection{Rendering optimizations}** \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**

Mobx package allows to perform the optimizations in an easy and direct way. All the effort is taken by the packages itself and the only thing to be cared about is to fragment the state in the most suited granularity for the purpose. In other words it is sufficient to make the state observable in the right points and wrap the corresponding UI elements into Observer widgets to get the job done. In the implementation provided so far for the list of todos, the only observable part was the list itself. The list is observed by the package and by the Observer widget contained in the TodoView. There is no concept of observability in the single todos yet and , once an instance of todo is change, what is seen by the package is just a completely new list. Consequently , every Observer widget listening for changes in the todo list is rebuilt. The smallest observable element the package sees is the list itself. Said that, the first thing to do in order to optimize the renderings is to increase the granularity of the observed state. Not the list only but also every contained todo needs to be made observable. It is necessary tough to redefine the todo model. This necessity arises with the usage of the mobx package and has not been faced in the other state management solutions used so far. The mobx state management solution introduces a dependency also in the data layer and in the model definition beside the usual dependency introduced in the business logic layer. In some scenarios this could represent an untoward drawback, for example in the porting processes. The Todo model class need to be made abstract an to implement the Store behaviour as we did for the TodoStore. Now it is possible to annotate its internal fields with the @action syntax. All the fields except the id need to be observable. Moreover, the final attribute must be removed to all the observed variables. It is indeed meaningless to observe a variable that cannot be mutated.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}  
part 'todo.g.dart';  
  
  
class Todo = \_Todo with \_$Todo;

abstract class \_Todo with Store{  
  
 final int id;  
 @observable  
 String name;  
 @observable  
 String description;  
 @observable  
 bool completed;

(…)

}

\end{minted}

\mbox{}

\end{code}

Using the line of code presented HERE RIFERIMENTO the todo.g.dart file is generated. The TodoItem widget is “stateless” for the moment. It receives a copy of the corresponding todo instance from the parent. We need to retrieve the instance directly from the store if we want the Observer widget to rebuild correctly and react to changes. Instead of receiving the entire todo from the parent the id field is enough. It is used then in the build method to access the corresponding todo instance in the store using the Provider widget. The remaining TodoItem code remains the same except for the fact that it is wrapped into an Observer widget.

\begin{code}

\mbox{}\\

\captionof{listing}{Todo app - InheritedWidget - extension to InheritedWidget} \mbox{}

**\label{code:2.14}**

\begin{minted}{dart}

class TodoItem extends StatelessWidget {  
 final int id;  
  
 const TodoItem({Key? key, required this.id}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 final store = Provider.*of*<TodoStore>(context, listen: false);  
 final todo = store.todos.where((element) => element.id == id).first;  
 return Observer(builder: (\_) {  
 print("building TodoItem ${todo.id}");  
  
 return InkWell(…);  
 });  
 }  
}

\end{minted}

\mbox{}

\end{code}

**\subsubsection{Conclusions}** \mbox{}\\ **\label{par:todo\_app\_inherited\_widget\_introduction}**

**The @readonly “issue”**

**We have seen that MobX allows to enable the strict mode making state changes possible only using actions. However**

# The other app

# Comparisons

# Conclusions