POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Mobile applications State Management in Flutter

Tesi di Laurea Magistrale in
Computer Science - Ingegneria Informatica

Author: **Lorenzo Ventura**

Student ID: 906003
Advisor: Prof. Luciano Baresi
Co-advisors:
Academic Year: 2021-2022

Abstract

Abstract

Keywords: here, the keywords, of your thesis

Abstract in lingua italiana

Qui va l'Abstract in lingua italiana della tesi seguito dalla lista di parole chiave.

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

Introduction

Introduction

1 | State management solutions

here i will present some main concepts and functionalities of the state management solutions proposed. This chapter will be filled with the information contained in the other word file i sent you.

1.1. SetState and InheritedWidget/InheritedModel

...

1.2. Redux

...

1.3. BLoc

...a

1.4. MobX

...

1.5. GetX

...

2 | The Todo app

This chapter is devoted to the implementation of a mobile application regarding the management of a number of todos. It is developed using the state managemenent solutions proposed in Chapter 1. For every solution, three different development processes are carried out. Moreover, a series of measurements ,concerning the volume of the code and the effort spent, will be collected.

2.1. General overview

This section explains in details the three development processes and the resulting application. These processes consist in the implementation of the main functionalities, in the addition of new ones and in some performance optimizations.

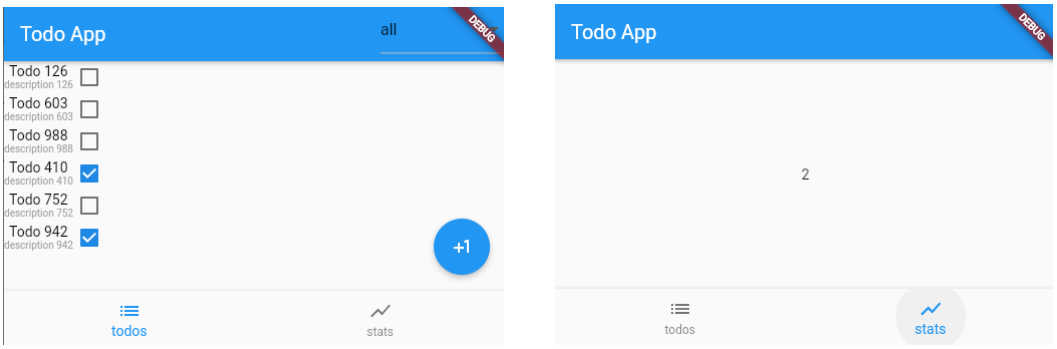
2.1.1. Base functionalities

This part of the development process aims to realize the skeleton of the application and the main functionalities. The output of the process will be and application which offers the possibility to visualize and partially handle todos. It is made of a single page: the HomePage. The HomePage is composed by an AppBar and two tabs: the *todos* tab and the *stats* tab.
In the *todos* tab the list of todos is visualized. Is possible to filter todos using a DropdownButton widget in the top right corner, inside the AppBar. The available filter values are:

- All (visualizes completed and pending todos)
- Completed (visualizes completed todo only)
- Not Completed (visualizes pending todos only)

The list of todos is visualized using a TodoView component widget. The elements that compose the list of todos are called TodoItems. TodoItem widgets visualize the todo's name and description using two Text widgets and the completion using a Checkbox widget.

It is possible to use the checkbox to mark a todo as completed or to mark it as pending depending on its current state. In the *stats* tab is possible to visualize the number of completed todos through a Text widget. In the lower part a TabSelector allow to switch from tabs.



(a) *todos* tab runtime UI (b) *stats* tab runtime UI.

Figure 2.1: Shows the HomePage UI

2.1.2. Adding new features

This part of the development process aims to add two new features to the output application of the previous step. This process is divided into two subprocesses. Both of them aims to add a single new feature.

The Add todo Feature - The first subproces adds the possibility to create new todos. It utilizes the FloatingActionButton widget, already present in the bottom right corner of the application, to push a new page called: AddTodoPage. In the AddTodoPage is possible to compile two TextField widgets and utilize a TextButton widget to return to the HomePage creating the new todo.

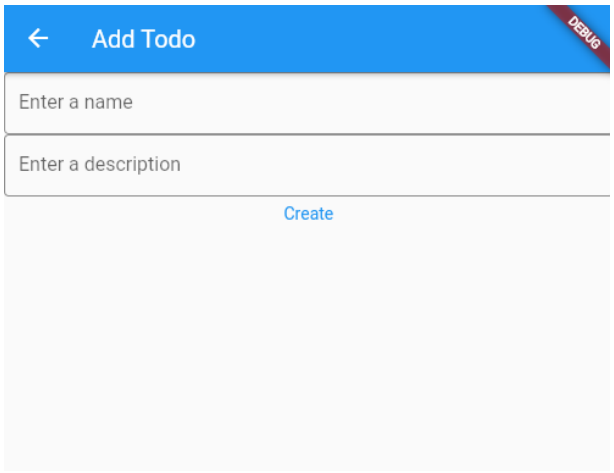


Figure 2.2: Shows the AddTodoPage UI

The Update feature - The second subproces aims to add the possibility to update existing todos. Tapping on a specific TodoItem allow to navigate to a new page : the UpdateTodoPage. In the UpdateTodoPage is possible to compile two TextFields widgets and use a TextButton widget to return to the HomePage modifing the corresponding todo.

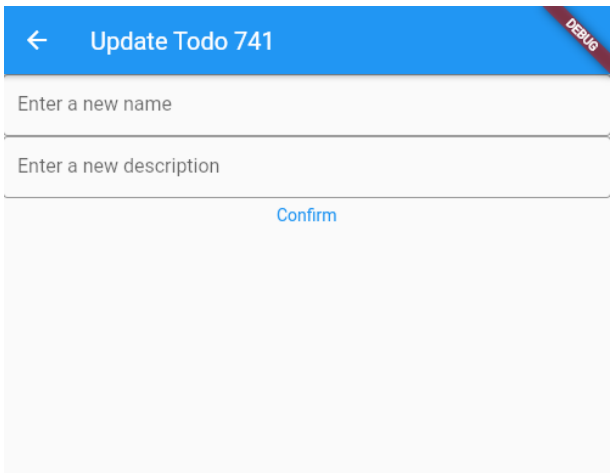


Figure 2.3: Shows the UpdateTodoPage UI for the todo with id 741

2.1.3. Renders optimization

This part of the development process aims to perform some optimizations in terms of UI renderings and memory consumption. In particular, the code will be refactored in order to use the least UI renders possible or ,in other words, to call the least *build* method runs possible. The focus is on the TodoView and TodoItem widgets. The TodoView widget

should be rendered again only after a structural change in the *filteredTodos* list, where the filteredTodos list represent the list containing the todos matching the current filter. A structural change is intended as a mutation of the length of the list or a substitution of its internal elements. Basically, a structural change occurs when a new todo is added or removed from the list or when the filter changes. If the filteredTodos list's change concerns a single todo (e.g. when its internal state is changed using the checkbox or the update feature) it is considered a non-structural change. The main difference is that, a structural change, needs to rebuild the entire TodoView widget ,instead , a non-structural change can rebuild only a subpart (the particular TodoItem widget). When a structural change occurs, more than one TodoItem widget is affected and by the change ,the most convinient way to mutate them all consistently ,is to rebuild the entire TodoView widget. Moreover, adding , deleting and substituting a TodoItem (and consequently add/delete/substitute a child to the TodoView tree's node) is only feasible by the parent widget and not by widgets on the same tree level. A non-structural change ,instead, affects only a specific TodoItem/child and so, it is possibile to rebuild the single element only. Those optimizations are not really necessary in this scenario. The implemented application is ,indeed, very simple and do not need this kind of improvements at all. This is just an experiment in order to define which solution performs better at handling optimizations and to give an adjunctive prospective in the final comparison.

2.2. Implementation

This section contains the implementation of the application presented in the Section 2.1.

2.2.1. Shared project structure and files

In order to make comparisons fairer , the code about the application's core and the UI is shared between different solution's implementations. In this way , measurements are taken on the part of the code added to the application by solution only. This subsection presents the shared code in details. Some parts of the shared code can change from one implementation to another in order to adapt to the solution. However, changes to this structure are kept minimal. The same is for the UI. It uses the least widgets and visual elements possible. In the Figure 2.4 the shared folder's and file's structure is shown. Subsequent paragraphs exaplain how models, pages, components and the repository are implemented.

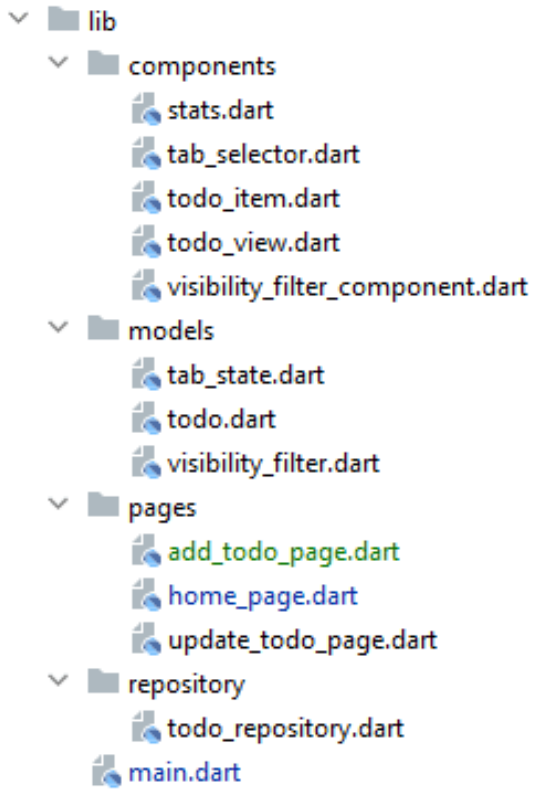


Figure 2.4: Todos app's shared files structure

The application's Root - The root widget of the application is called MyApp. It is a stateless widget composed by a MaterialApp widget. Inside the MaterialApp widget, three routes are defined : the HomePage , the UpdateTodoPage and the AddTodoPage. The *inizialRoute* is set to the HomePage as deafult. The UpdateTodoPage takes a Todo instance as argument. Inside the *main* function the MyApp widget is passed to the *runApp* method in order to start the application.

Source Code 2.1: Todo app - MaterialApp and main function definition

```
void main() {
  //launching the application
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      initialRoute: "/", //setting initial route to the HomePage
      routes: { //defining possible routes
        "/": (context) => const HomePage(),
        "/updateTodo": (context) => UpdateTodoPage(
              todo: (ModalRoute.of(context)!.settings.arguments
                      as Todo),
        "/addTodo": (context) => AddTodoPage(),
      },
    );
  }
}
```

Models - The model for the HomePage's tabs is implemented using an enumeration.

Source Code 2.2: Todo app - TabState model definition

```
enum TabState{
        todos,stats
}
```

Filters for the *filteredTodos* list are modelled by an enumeration too. They can take three values: *all, notCompleted, completed.*

Source Code 2.3: Todo app - VisibilityFilter model definition

```
enum VisibilityFilter{
        completed,notCompleted,all
}
```

It's not possible to give a common implementation for the Todo's model matching every solution. Todo's model, indeed, can change in different implementations. The sharable structure of the model, however, can defined as follows. (

Source Code 2.4: Todo app - Todo model definition

```
        @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;
}
```

Repository and utilities - Some usefull functions are shared between different implementations. They are contained in the utility.dart file. *generateId* function takes a list of todos and generate a new unique id. *todoExists* function takes a list of todos and a id and check if a todo with that id exists.

Source Code 2.5: Todo app - TodoRepository definition

```
int generateId(List<Todo> todos) {
```

```
  Random rand = Random();
  int newInt = rand.nextInt(1000) + 2;
  List<int> ids = todos.map((todo) => todo.id).toList();
  while (ids.contains(newInt)) {
    newInt = rand.nextInt(1000) + 2;
  }
  return newInt;
}

bool todoExists(List<Todo> todos, int id) {
  List<Todo> result = todos.where((todo) => todo.id == id).toList();
  return result.isNotEmpty ? true : false;
}
```

The TodoRepository class simulates todos's fetching process from a Database. It has two static methods. These methods are asynchronous and have a duration of 2 seconds to give the impression of a real asynchronous operation. The method *loadTodos*, in particular, populates a list with six new todo instances, using the *generateId* function for the generation of their unique IDs. Subsequently, 2 seconds later, it returns the list to the caller.

Source Code 2.6: Todo app - TodoRepository definition

```
class TodoRepository {
  //generate 6 unique todos
  static Future<List<Todo>> loadTodos() async {
    List<Todo> todos = [];
    Random rand = Random();
    while (todos.length < 6) {
      int newInt = generateId(todos);
      todos.add(Todo(
          id: newInt,
          name: "Todo " + newInt.toString(),
          description: "description " + newInt.toString(),
          completed: rand.nextBool()));
    }
//waiting 2 seconds before returning the list
```

```
    await Future.delayed(const Duration(seconds: 2));
    return todos;
  }

  static Future<void> saveTodos(List<Todo> todos) async {
    await Future.delayed(const Duration(seconds: 2));
  }
}
```

Components - Components are widgets created with a specific task. They provide some sort of indipendent functionality. TodoView widget component, for example, takes care of visualizing a list of todos. Todos are accessed in different ways depending on the implementation. TodoView widget uses a ListView widget filled with TodoItem widgets. *itemCount* and *itemBuilder* fields are left empty for future implementation.

Source Code 2.7: Todo app - TodoView definition

```
class TodoView extends StatelessWidget {

  const TodoView({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building TodoView");

    return ListView.builder(
      itemCount: //to be filled,
      itemBuilder: (context, index) {
        return TodoItem(
        todo: //to be filled
        );
      },
    );
  }
}
```

TodoItem widget component takas care of visualizing a specific todo. TodoItem widget is a stateless. It uses two Text widgets in order to display the todo's information and a

Checkbox widget to allow the changing of the todo's completed field. It is wrapped into an InkWell widget to make it responsive to taps. Functions fields are left empty for future implementation.

Source Code 2.8: Todo app - TodoItem definition

```
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" , arguments: todo);
      },
      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) {
              //to be filled
              }),
        ],
      ),
```

```
            );
          }
        }
```

TabSelector widget component provides the tab switching feature. It uses a BottomNavigationBar widget filled with as many BottomNavigationBarItem widgets as TabState.values (in our case two). Functions fields are left empty for future implementation.

Source Code 2.9: Todo app - TabSelector definition

```
class TabSelector extends StatelessWidget {

  const TabSelector(
      {Key? Key})
      : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building Tab Selector");

    return BottomNavigationBar(
      currentIndex: //to be filled ,
      onTap: (){
      //to be filled
      },
      items: TabState.values
          .map((tab) => BottomNavigationBarItem(
                label: describeEnum(tab),
                icon: Icon(
                  tab == TabState.todos ? Icons.list : Icons.show_chart,
                ),
              ))
          .toList(),
    );
  }
}
```

VisibilityFilterSelector widget component uses a DropdownButton widget filled with as many DropdownMenuItems widget as VisibilityFilter.values (in our case three). Function fields are left empty for future implementation.

Source Code 2.10: Todo app - VisibilityFilterSelector definition

```
class VisibilityFilterComponent extends StatelessWidget {

  const VisibilityFilterComponent(
      {Key? key})
      : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building Visibility filter");

    return DropdownButton<VisibilityFilter>(
      value: //to be filled,
      items: VisibilityFilter.values.map((filter) {
        return DropdownMenuItem<VisibilityFilter>(
            child: Text(describeEnum(filter)), value: filter);
      }).toList(),
      onChanged: (filter) {
       //to be filled
      },
    );
  }
}
```

Stats widget component takes care of visualizing some numerical representation regarding the list of todos. Stats component is a stateless widget composed by a Text widget, showing the stats value, wrapped into a Center widget.

Source Code 2.11: Todo app - Stats definition

```
class Stats extends StatelessWidget {
```

```
  const Stats({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building Stats");

    return Center(
        child: Text(
        // to be filled
        ));
  }
}
```

Pages - The HomePage can be a statefull widget or a stateless widget depending on the utilized solution. In both cases it uses a simple Scaffold widget. The AppBar widget contains a VisibilityFilterSelector only when the tab is set to *todos*. The body is filled with a TodoView component when the tab is set to *todos* and with a Stats component when the tab is set to *stats*. The body can change from *todos* tab to *stats* tab using the BottomNaviagationBar (filled with the TabSelector widget). An empty FloatingActionButton is also present for future implementation. (note: some small pieces could change in different solution's implementation. In the above example the tab value is contained in the HomePage but it will not be always the case).

Source Code 2.12: Todo app - HomePage definition

```
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 TodoProvider(
      child: Builder(builder: (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(),
          floatingActionButton: tab == TabState.todos
              ? FloatingActionButton(
                  child: const Icon(Icons.plus_one),
            onPressed: () {
              Navigator.pushNamed(context, "/addTodo");

            },
          ) : Container(),
        );
      }),
    );
  }
}
```

The UpdateTodoPage is statefull and uses a Scaffold widget too. The reason it is implemented with a statefull widget is that it uses TextEditingController objects that need a statefull widget to work properly. The body is filled with a Column with two TextField widgets and a TextButton widget. The TextButton is left empty for future implementation.

Source Code 2.13: Todo app - UpdatePage definition

```
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 textControllerName = TextEditingController();
  final textControllerDesc = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Update Todo" + widget.todo.name),
        ),
        body: Column(
          children: [
            TextField(
              controller: textControllerName,
              decoration: const InputDecoration(
                  border: OutlineInputBorder(),
                  hintText: 'Enter a new name'),
            ),
            TextField(
              controller: textControllerDesc,
              decoration: const InputDecoration(
                  border: OutlineInputBorder(),
                  hintText: 'Enter a new description'),
            ),
            TextButton(
                onPressed: () {
                  //to be filled
                },
                child: const Text("Confirm"))
```

```
          ],
        ));
  }

  @override
  void dispose() {
    textControllerName.dispose();
    textControllerDesc.dispose();
    super.dispose();
  }
}
```

The AddTodoPage is statefull too and uses a Scaffold widget. The body is filled with a Column widget with two TextField widgets and a TextButton widget inside. The TextButton *onChanged* field is left empty for future implementation.

Source Code 2.14: Todo app - AddTodoPage definition

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

  const AddTodoPage({Key? key, required this.addTodoCallback})
      : super(key: key);

  @override
  State<AddTodoPage> createState() => _AddTodoPageState();
}

class _AddTodoPageState 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: () {
                  //to be filled
                },
                child: const Text("Create"))
          ],
        ));
  }

  @override
  void dispose() {
    textControllerName.dispose();
    textControllerDesc.dispose();
    super.dispose();
  }
}
```

Here ends the implementation of the shared files.

2.2.2. InheritedWidget/Model and SetState implementation

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

Base functionalities

Here starts the implementation of the base functionalities exposed in Subsection 2.1.1.

Core state - In order to use InheritedWidget's functionalities, a new class must be defined and extended with InheritedWidget class. For our purpose, a single class will be enough to contain all the application's state. This new class is called *TodoInheritedData*.

Source Code 2.15: Todo app - InheritedWidget - extension to InheritedWidget

```
class TodoInheritedData extends InheritedWidget{
```

The application's state is composed by: a list of Todos, a VisibilityFilter , a Int for the stats (for conciseness it will represent the number of completed todos) and filtered list of todos that contains the todos matching the filter. Inside the constructor, final variables are initialized with their corresponding arguments and, *stats* and *filteredTodos* list, are computed.

Source Code 2.16: Todo app - InheritedWidget- TodoInheritedData implementation

```
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, //computing stats
      filteredTodos = filterTodo(todos, filter),//computing filtered list
      super(child: child, key: key);
}
```

filterTodos function is just a function that takes a list of todos and a visibility filter and returns the filtered list. Important to notice is the fact that a *child* widget must be also provided in the constructor. This is because TodoInheritedData is nothing else than a widget itself that wraps data and makes it accessible down the tree.

TodoInheritedData widget is stateless. It cannot be changed (every value is final) , instead, a new TodoInheritedData widget must be provided when a data change occurs. The *updateShouldNotify* method must be overridden inside the TodoInheritedData class. This method belongs to the InheritedWidget class and its override is mandatory. It helps to avoid useless UI rebuilding when a new state, without actual data changes, occurs. Once a TodoInheritedData element is replaced with a new one, the new element takes care of calling the *updateShouldNotify* method and decides whether is necessary to notify changes in the subtree. If the method returns *true* , the subtree is rebuilt, if it returns *false*, instead, it is not.

Source Code 2.17: Todo app - InheritedWidget -updateShouldNotify method override

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

listEquals function is provided by Dart language. It takes two lists and compares them element by element, returning true if all elements are equal. In the source code RIFEIRMENTO, it takes as parameters the old *filteredTodos* list (the one belonging to the old widget) and the new filteredTodos list and compares them. In case no changes were performed it returns *true* and leads the *updateShouldNotify* function to return *false*, leaving the subtree unchanged.
InheritedWidget class requires also the *of* method override. The *of* method makes the instance of the TodoInheritedData class accessible down the tree. It is a static method (it can be called without istantiating any TodoInheritedData object) and returns the instance of the nearest TodoInheritedData widget up in the tree. It extracts the instance from the current *context* object using the method called *dependOnInheritedWidgetOfExactType* provided by the framework. In case no TodoInheritedData instance is found it raises a runtime error.

Source Code 2.18: Todo app - InheritedWidget - TodoInheritedData of method override

```
static TodoInheritedData? of(BuildContext context) {
 final TodoInheritedData? result =
  context.dependOnInheritedWidgetOfExactType<TodoInheritedData>();
assert(result != null, 'No TodoInheritedData found in context');
return result;
}
```

TodoInheritedData widget is now ready to be used. In the overall it is a container for our state. It makes the state accessible in the subtree but, is not clear yet who is really filling it with the correct informations. TodoInheritedData widget represents the state of the appplication in a given moment. It cannot change its internal values neither substitute itself with another instance. In practice , what happends, is that a stateful widget is created. This stateful widget contains the state and bothers to create a new instance of the TodoInheritedData widget every time the state changes. Everytime its internal state is changed (using *setState*), indeed, a new instance of TodoInheritedData widget is produced and substituted with the old one. In this way, changes are reported to the subtree which sees a different image of the state and reacts to it. I personally did not appreciate this way of doing InheritedWidget uses. On one way it is simple and works really well for its purpose , on the other hand it introduces a new level of data caching. The concept of data caching will be explained a bit more in details later but ,for the moment ,we can say that the application's state is not exactly atomic/unique. What is seen by the subtree is a screenshot of the state and not the state itself. The real state is contained in the stateful widget. It is important, though, that the real state and the screenshot provided in the subtree are well syncronized. A bad syncronization can produce inconsistency in what is visualized and the information contained in the internal state. More in general, it can be said , that the more data caching level are introduced the harder it gets to efficiently syncronize them. It is clear that ,in our scenario ,this problem does not really show up. Or better, it will in the optimization part but ,in that case, InheritedWidget tool is used with a purpose that goes behoind its real usage. Anyway, it is possible that different widgets sees different screenshots of the data and the bigger the application grows the higher will be the probability that this case show up. Now that the background is a bit clearer the implementation process can continue. As mentioned above, a new stateful widget must be created. This new stateful widget is called *TodoProvider*. It has two variables representing the state: a list of todos and a filter. (the rest of the state is computed at each TodoInheritedData creation)

Source Code 2.19: Todo app - InheritedWidget - TodoProvider implementation

```
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 is set as *all* by default. In the statefull widget's *init* method , todos are fetched from the repository and pushed inside the *todos* variable using the *setState* method.

Source Code 2.20: Todo app - InheritedWidget - TodoProvider 's init method implementation

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

```
  });
  super.initState();
}
```

At this point, our TodoProvider widget can be incorporated as the parent of the Scaffold widget in the HomePage. The usage of the Builder widget is due to the fact that the instante of TodoInheritedData is accessible only in a context where a TodoProvider is already present. In other words, TodoProvider's data cannot be accessed 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 containing the TodoProvider widget.

Source Code 2.21: Todo app - InheritedWidget - Data injection in the HomePage's subtree

```
class _HomePageState extends State<HomePage> {
  TabState tab = TabState.todos;

  @override
  Widget build(BuildContext context) {
    return TodoProvider(// new TodoProvider widget
      child: Builder(// new Builder widget
      builder: (context) {
        return Scaffold(...) // the rest of the HomePage;
        }
      ),
    );
  }
}
```

The TodoView component - TodoView component can now be populated. It is a stateless widget that looks up for the *filteredTodos* list, contained in the TodoInherited-Data widget. It uses the *of* method ,defined here RIFERIMENTO, to access the nearest TodoInheritedData instance. Then, it uses the list to populate the ListView widget.

Source Code 2.22: Todo app - InheritedWidget - TodoView implementation

```
class TodoView extends StatelessWidget {

  const TodoView({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building TodoView");
//retrieving the filtered list from the state
    final List<Todo> filteredTodos = TodoInheritedData.of(context).filteredTodos;

    return ListView.builder(
      itemCount: filteredTodos.length,// to use it here
      itemBuilder: (context, index) {
        return TodoItem(
          todo: filteredTodos.elementAt(index),// and here
        );
      },
    );
  }
}
```

The VisibilityFilterSelector component - At this point we got a single page (Homepage) that uses a TodoView widget to show the *filteredTodos* list contained in the TodoInheritedData widget. When the application starts , and empty page appears (todo are empty at the beginning) and then ,after few seconds , a list of todos, with their names, descriptions and completions, is shown. The list of filtered todos can be visualized , but is not interactable yet. In the HomePage's AppBar, a VisibilityFilterSelector component is ready to be used as defined RIFERIMENTO. Its current DropdownButton's *value* field is filled looking up for the *filter* value in the TodoInheritedData widget. Then, the *items* field is filled with a list of DropdownMenuItem widgets that comes from the mapping of all possible VisibilityFilter values to DropdownMenuItem widgets.

Source Code 2.23: Todo app - InheritedWidget - VisibilityFilterComponent implementation

```
class VisibilityFilterComponent extends StatelessWidget {
```

```
  const VisibilityFilterComponent(
      {Key? key})
      : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building Visibility filter");
    // retrieving the visibility filter from the state
    VisibilityFilter filter= TodoInheritedData.of(context).filter;
    return DropdownButton<VisibilityFilter>(
      value: filter, // use it here
      items: VisibilityFilter.values.map((filter) {
        return DropdownMenuItem<VisibilityFilter>(
            child: Text(describeEnum(filter)), value: filter);
      }).toList(),
      onChanged: (filter) {
        //to be implemented
      },
    );
  }
}
```

The *onChanged* field must be populated with a function that takes as single argument a VisibilityFilter value. This function is called when a DropdownMenuItem is tapped by the user. It contains ,as argument, the tapped DropdownMenuItem's filter value. We want this function to change the state contained in the TodoInheritedData widget (the *filter* variable) when fired. In order to do so , a state changing function must be provided, by the TodoInheritedData widget, to be accessed ,and called, by widgets. As we mentioned earlier, TodoInheritedData widget contains only final fields and should never be modified. It is not possible ,indeed, to directly change the values inside the TodoInheritedData widget. For this reason , just adding a new function inside the TodoInheritedData widget ,to perform the change, is not a solution. Indeed, trying to change a part of the state, inside this ipotetic function, will generate an error at compile time (final variable cannot be set outside constructor). A completly new TodoInheritedData widget ,indeed, should be created. The TodoInheritedData widget is created in the TodoProvider widget, when the *build* method runs, using its local variables *todos* and *filter*. In order to generate a new

TodoInheritedData widget, is sufficient to change the TodoProvider widget's local state ,using the *setState* method. This will cause the *build* method to run again with the new values and to generate a new TodoInheritedData widget. At this point should be clear that the state changing function comes from the TodoProvider widget. This function , once called, changes the local state of the TodoProvider stateful widget generating a new state for the application.
In pratice, a new function, called *onChangeFilter*, is added inside the TodoProvider widget. This function takes a VisibilityFilter value as parameter and sets the value of TodoProvider's *filter* variable using the *setState* method.

Source Code 2.24: Todo app - InheritedWidget - TodoProvider's onChangeFilter implementation

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

Once called, being the state (the part concerning the filter) changed, another run of the *build* method is performed. As a conseguence the TodoInheritedData widget ,present in the tree, is replaced with the new one. However, widgets access the state through the TodoInheritedData widget and not through the TodoProvider widget. For this reason, an instance of the *onChangeFilter* function must be provided to the TodoInheritedData widget to make it accessible in the subtree. A new parameter is added, tough , in the TodoInheritedData class.

Source Code 2.25: Todo app - InheritedWidget - TodoInheritedData widget expansion

```
class TodoInheritedData extends InheritedWidget {
  final List<Todo> todos;
  final List<Todo> filteredTodos;
  final void Function(VisibilityFilter) onChangeFilter; //new variable
  final int stats;
  final VisibilityFilter filter;
```

The *onChangeFilter* function is then passed to the TodoInheritedData widget on its creation.

Source Code 2.26: Todo app - InheritedWidget -onChangeFilter function injection into TodoInheritedData widget

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

Now that the *onChangeFilter* function is accessible down in the tree, it can be called in the *onChange* field of the DropdownButton widget, inside the VisibilityFilterSelector component.

Source Code 2.27: Todo app - InheritedWidget - DropdownButton's onChanged field implementation

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

It is now possible to apply different filters to the list of todos in the Homepage.

The TodoItem component - TodoItem widget is stateless for the moment. It takes as paramenter a Todo instance and takes care of displaing it. TodoItem widget does not access the state. The todo to be displayed is, indeed, passed by the parent widget (the TodoView). However, the TodoItem widget needs to write the state once the checkbox is tapped. For the moment, the Checkbox widget is just showing the value of the completed field and Its *onChange* function is still empty. When the checkbox is tapped, a change into the corresponding Todo's *completed* field should be fired and a rebuild of

the TodoItem widget performed. In order to do so, TodoIhneritedData widget should provide a state changing function for the completed field. The process to be done is the same exposed in the previous paragraph. Into the TodoProvider stateful widget a new function ,called *onSetCompleted* , is created. This function takes as parameter the *id* of the todo to be changed and the new value for the *completed* field.

Source Code 2.28: Todo app - InheritedWidget - TodoProvider widget *onSetCompleted* function implementation

```
void onSetCompleted(int id, bool completed) {
    //control the todo's existance
    assert(todoExists(todos, id) == true, 'No todo with id : \$id');
    //change the state
    setState(() {
      todos = todos.map((todo) {
        if (todo.id == id) {
          return Todo(
              id: id,
              name: todo.name,
              description: todo.description,
              completed: completed);
        } else {
          return todo;
        }
      }).toList();
    });
  }
```

In the *onSetCompleted* function , *todos* list is scanned using the *map* method. Once the searched todo is found, its *completed* value is updated. Calling the *onSetCompleted* method on the TodoProvider stateful widget causes the *build* method to run again and to substitute the current TodoInheritedData widget with a new , updated, one. As before, the function is passed from the TodoProvider widget to the TodoInheritedData widget ,on its creation. In this way, the function is made accessible down the tree. It is now possible to call *onSetCompleted* function inside the *onChanged* field of the TodoItem's Checkbox.

Source Code 2.29: Todo app - InheritedWidget - TodoItem's Checkbox *onChanged* field implementation

```
Checkbox(
value: todo.completed,
    onChanged: (value) {
    //call the onSetCompleted method
    TodoInheritedData.of(context).onSetCompleted(todo.id, value!);
}),
```

At this point is possible to visualize the *filteredTodos* list, change the filter and update Todo's *completed* field.

The Stats component - Stats widget is stateless. It just needs to read the part of the state concerning the stats. The nearest instance of the TodoInheritedData widget is retrieved using the *of* method and used to fill in the Text widget.

Source Code 2.30: Todo app - InheritedWidget - Stats component implementation

```
class Stats extends StatelessWidget {
  const Stats({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("Building Stats");

    return Center(
        child:
        //retrive the value of the stats
        Text(TodoInheritedData.of(context).stats.toString()));
  }
}
```

The TabSelector component - The part of the state concerning the tab just includes one variable and is related to the HomePage only. The fact that a state management solution is being used to handle the application's state does not mean that it has to be used to handle everything. An important aspect of the state management in medium-

large applications is that, the core objective, still remains to handle things in the easiest way possible, as long as it works fine. There is no meaning in overcomplicating procedures that are easy to implement. Sometimes, indeed, for the parts of the state that can be refered as "local" ,meaning that they are relative to a small part of the application only, is not necessary to use complicated state management solutions. It is better to keep things simple and use the tools that most adapt to the specific scenario. In our case, there are two ways to implement the TabSelector widget: use *setState* and stateful widgets or use InheritedWidgets. The simpler one is to use setState as proposed RIFERIMENTO for more than one aspect. First, it is a good practice to keep the global state of the application as small and clean as possible. The bigger and the more complicated it gets the messier becomes to avoid bugs. Second, it is simpler , in pratice, to create a local variable and to hadle it with *setState* and stateful widgets instead of adding a new variable to the TodoInheritedData widget , manage it using the to TodoProvider widget and access it in the HomePage. The HomePage is already a stateful widget and is built using the *tab* variable. It is sufficient to add a new tab changing function to accomplish the task. This new function, called *onTabChange*, takes a int value as parameter and uses the *setState* method to update the *tab* variabile. This int value refers to the index of the tapped element inside the list of BottomNavigationBarItem widgets provided to the *items* field.

Source Code 2.31: Todo app - InheritedWidget - HomePage's onTabChange function implementation

```
TabState tab = TabState.todos;

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

However, the function *onTabChange* ,needs to be called in the TabSelector widget (and not in the HomePage). The easiest way is to pass the function, togheter with the tab variable, as parameter and use it in the BottomNavigationBar's *onTap* field.

Source Code 2.32: Todo app - InheritedWidget - TabSelector component implementation

```
class TabSelector extends StatelessWidget {
  final TabState currTab; // new parameter
  final Function(int) onTabChange; // new parameter

  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), //used here
      onTap: onTabChange, //used here
      items: (...)
    );
  }
}
```

It is now possible to switch from tabs. All the baase functionalities were implemented. The whole process was fast and straight forward.

Features addition

Here stats the development process which aims to add the new feature proposed HER RIFERIMENTO.

Todo addition feature - TodoInheritedData widget must provide a way in order to add todos to the list. A new function is created in the TodoProvider widget and passed to the TodoInheritedData widget as parameter. This new function is called *onAddTodo* and takes two parameters (name and description).

Source Code 2.33: Todo app - InheritedWidget - TodoProvider *onAddTodo* function implementation

```
  void onAddTodo(String name, String desc) {
    //generate a new unique id
    int newId = generateId(todos);
    //create the new todo
    Todo newTodo = Todo(
        id: newId,
        name: name,
        description: desc + " " + newId.toString(),
        completed: false);
    //perform the state change
    List<Todo> newList = List.from(todos);
    newList.add(newTodo);
    setState(() {
      todos = newList;
    });
  }
```

After generating a new unique id, it creates a new Todo instance called *newTodo* with the *completed* field set to *false* . Adding the *newTodo* to the TodoProvider's state *todos* list requires a bit of workaround. The state of a stateful widget is immutable. It is a bit counterintuitive but, as we already discussed, in functional programming is generally better to create new instances instead of mutating existing ones. Stateful widget's state can only be changed by the *setState* method. Unfortunately, the method *add* for lists, dart languege provides, is of type void and do not return a new list but adds the new value to the existing list's instance. For this reason, directly calling the *add* method to the TodoProvider's local lists *todos* would have no effect. That list is immutable and cannot be changed. TodoProvider's *todos* list must be completely replaced with a new instance containing the new todo. First, a new temporary list, called *newList*, is created and populated with the elements present in the *todos* list. Then, the *newTodo* is added to this new list. At this point, is sufficient to replace the *todos* list with the new one inside the *setState* method.
To make the *onAddTodo* function accessible down the tree, is sufficient to add a new field in the TodoInheritedData widget and pass the function to it, on its creation.

Source Code 2.34: Todo app - InheritedWidget - onAddTodo function propagation

```
class TodoInheritedData extends InheritedWidget {
```

```
  final void Function(String,String) onAddTodo; // new variable
  final List<Todo> todos;
  final List<Todo> filteredTodos;
  final void Function(VisibilityFilter) onChangeFilter;
  final int stats;
  final VisibilityFilter filter;

  (...)

class TodoProvider extends StatefulWidget {
@override
Widget build(BuildContext context) {
  return TodoInheritedData(
    todos: todos,
    onChangeFilter: onChangeFilter,
    onAddTodo: onAddTodo, // passing the onAddTodo function
    onSetCompleted: onSetCompleted,
    filter: filter,
    child: widget.child,
  );
}
```

In the AddTodoPage, a TextButton widget has been already set up and is ready to call the *onAddTodo* function once tapped. However, there is a small inconvenient. The AddTodoPage is accessed by pushing on top of the HomePage another route as shown in figure 2.20. The AddTodoPage is added as a child of the MaterialApp widget.

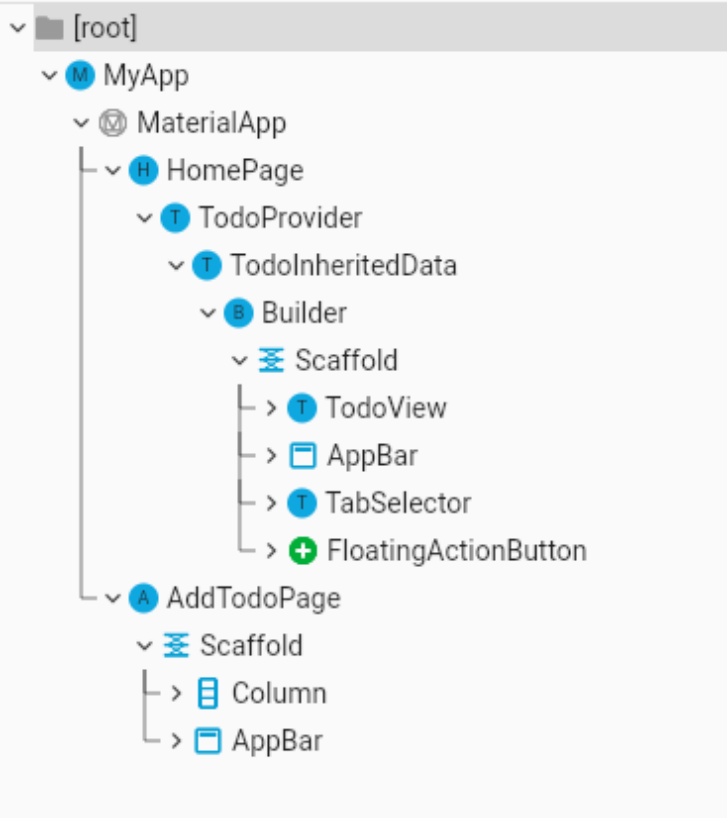


Figure 2.5: Shows the widgets tree structure in the AddTodoPage

The AddTodoPage is not a part of the subtree of the HomePage but is a standalone tree. There is no instance of TodoProvider widget as ancestor of the Scaffold widget present in the AddTodoPage. It is not possible, tough, to call the *of* method as we did before. Indeed, calling the *of* method in a context where a TodoProvider widget is not present will cause the line

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

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

Source Code 2.35: Todo app - InheritedWidget - AddTodoPage's callback function parameter creation

```
class AddTodoPage extends StatefulWidget {
  final void Function(String, String) addTodoCallback; // new parameter
```

```
  const AddTodoPage({Key? key, required this.addTodoCallback})
      : super(key: key);
```

Then, the MaterialApp is notified about the necessity of this new argument at the AddTodoPage creation. It will find the argument inside the context, in a specific variable called *arguments*.

Source Code 2.36: Todo app - InheritedWidget - MaterialApp AddTodoPage route re-definition

```
routes: {
{. . .}
  "/addTodo": (context) => AddTodoPage(
  // passing the onAddTodo functio by argument
      addTodoCallback: ModalRoute.of(context)!.settings.arguments
          as Function(String, String)),
},
```

The callback function in then used in the *onPressed* field of the TextButton widget in the AddTodoPage. Once the TextButton is tapped, the new todo is added to the list and the AddTodoPage is popped returning to the HomePage. Being the state change the HomePage is rebuilt.

Source Code 2.37: Todo app - InheritedWidget - AddTodoPage TextButton onPressed field implementation

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

Raising the TodoProvider widget above the MaterialApp widget would not be a good solution. The higher the TodoProvider widget resides in the tree the more widgets are rebuilt on state changes. In this case it is easier to pass the callback function as parameter to the AddTodoPage.

Todo updating feature - A new function must be implemented in the TodoProvider widget and passed to the TodoInheritedData widget. This new function is called *onUpdateTodo* and takes three arguments: the *id* of the todo to be updated, the *newName* and the *newDesc*.

Source Code 2.38: Todo app - InheritedWidget - todoProvider's onUpdateTodo function implementation

```
void onUpdateTodo(int id, String newName, String newDesc) {
  //control the todo's existance
  assert(todoExists(todos, id) == true, 'No todo with id : \$id');
  //create a new list with the updated todo
  List<Todo> newTodosList = todos.map((todo) {
    if (todo.id == id) {
      return Todo(
          completed: todo.completed,
          description: newDesc,
          name: newName,
          id: todo.id);
    } else {
      return todo;
    }
  }).toList();
  //update the state
  setState(() {
    todos = newTodosList;
  });
}
```

The *onUpdateTodo* function checks if a todo matching the id exists. Then, for the same immutability concept we dealt with in the previous paragraph 2.30, a *newTodosList* is created and populated with the elements of the *todos* list. Moreover, the todo with the corresponding id is updated with the new name and the new description. Finally, the *todos* list in the TodoProvider stateful widget is replaced with the *newTodosList* using the *setState* method. This new *onUpdateTodo* method is then made accessible down the tree adding a field to the TodoInheritedData widget.

Source Code 2.39: Todo app - InheritedWidget - onUpdateTodo function propagation

```
class TodoInheritedData extends InheritedWidget {

  final void Function(int, String,String) onUpdateTodo; // new variable
  final void Function(String,String) onAddTodo;
  final List<Todo> todos;
  final List<Todo> filteredTodos;
  final void Function(VisibilityFilter) onChangeFilter;
  final int stats;
  final VisibilityFilter filter;

  (...)

class TodoProvider extends StatefulWidget {

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

For the same problem faced during the implementation of the todo addition 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 is of type Function taking two Strings as arguments (the id will be already set up by the calling page).

Source Code 2.40: Todo app - InheritedWidget - UpdateTodoPage callback variable creation

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

  const UpdateTodoPage({Key? key, required this.todo, required this.callback})
      : super(key: key);

 (...)
```

We are ready now to push the UpdateTodoPage on top of the Homepage when the InkWell widget (inside the TodoItem widget) is tapped. However, there is a small extra step to perform before proceeding. Flutter Navigator ,indeed, allows to pass a single object ,as argument ,between routes. In this case not only the *onUpdateTodo* function but also a instance of the Todo must be passed to the UpdateTodoPage. For this reason a wrapper class is created with the name *UpdateTodoPageArguments* .

Source Code 2.41: Todo app - InheritedWidget - onUpdateTodo function propagation

```
class UpdateTodoPageArguments {
  final Todo todo;
  final void Function(String ,String) updateState;

  UpdateTodoPageArguments({required this.todo, required this.updateState});
}
```

Inside the InkWell's *onTap* function ,the corresponding todo and the *onUpdate* function are wrapped into an object of type UpdateTodoPageArguments. This object is then passed to the new route.

Source Code 2.42: Todo app - InheritedWidget - onUpdateTodo function propagation

```
Navigator.pushNamed(context, "/updateTodo",
//passing the the state changing function using a wrapper class
```

```
    arguments: UpdateTodoPageArguments(
        todo: todo,
        updateState: (String newName,String newDesc) {
          TodoInheritedData.of(context, aspect: 0)
              .onUpdateTodo(todo.id, newName,newDesc);
        }));
```

It is necessary to specify to the MaterialApp widget where, the two parameters (necessary for the UpdateTodoPage creation), will be situated. As before , they are putted in a specific variable, inside the context object, called *arguments.*

Source Code 2.43: Todo app - InheritedWidget - onUpdateTodo function propagation

```
routes: {
  "/": (context) => const HomePage(),
  "/updateTodo": (context) => UpdateTodoPage(
        todo: (ModalRoute.of(context)!.settings.arguments
                as UpdateTodoPageArguments)
            .todo,
        callback:
//access the wrapper class to populate the callback parameter
         (ModalRoute.of(context)!.settings.arguments
                as UpdateTodoPageArguments)
            .updateState,
      ),
```

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

Source Code 2.44: Todo app - InheritedWidget - onUpdateTodo function propagation

```
TextButton(onPressed: () {
  //call the onUpdateTodo function
  widget.callback(textControllerName.text,textControllerDesc.text);
  Navigator.pop(context);
},
```

At this point, once the user taps the confirm button the page pops, the corresponding todo updates and the HomePage rebuilds.

Rendering optimizations

I spent some hours trying to figure out how to make the single TodoItem widget rebuild only, after a non -structural change occurs. When a non-structural change occurs, may be instresting, tough, to limitate the tree rebuilding to widgets affected by the mutation only. For example, when the Checkbox inside a TodoItem is tapped, would be nice to rebuild the TodoItem widget only, and not the entire TodoView widget. After some attempts, I realized that it was just not feasible using InheritedWidgets alone. InheritedWidgets, indeed, do not offer this possibility at all. Every widget that access the state in the TodoProvider's subtree, using the *of* method, is registered as *listener* for state changes. Once a state change occurs ,there are only two possibilities: notify all listeners and rebuild them or notify none. In other words when a state change occurs, and it must be visualized , the entire TodoProvider's subtree must be rebuilt unconditionally. Flutter framework, however, offers a particular widget, called InheritedModel, to handle this kind of scenario. InheritedModel works as InheritedWidget except for the fact that, when a widget accesses the state ,(calling the *of* method) it must provide also a new additional parameter, called *aspect*. The *aspect* parameter can be whatever object. For example a String or a Int, but also a more complex data structure. The *aspect* parameter identifies which part (or parts) of the state the widget is registering to. With this new additional tool is possible to achive the partial rendering we were looking for. Indeed, with InheritedModel , widgets are rebuilt based on the changed aspect of the state. If a particular widget registered for a particular aspect and a state mutation, not affecting that aspect, occurs, the widget is not rebuilt. However, the entire logic defining which aspect of the data changed (when a state transition occurs) must be implemented by the programmer.
The extension to InheritedWidget is substituted with the InheritedModel one, in the TodoInheritedData class.

Source Code 2.45: Todo app - InheritedModel - extension to InheritedModel

```
class TodoInheritedData extends InheritedModel<int> {
```

I decided to use Ints in order to identify aspects. In particular, widgets that need to rebuild on *filteredTodos* list structural change, will register to the aspect identified with the number 0. Widgets that do never need to rebuild will register to the aspect identified with number 1. Widgets that need to rebuild when a non-structural change occurs affecting the specific Todo with id n 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). At this point the method *of*, in the TodoInheritedData widget, should be updated taking into account the aspect parameter. Morevover, the *result* variable should be populated with the static *inheritedFrom* method belonging to the InheritedModel class, instead of the *dependOnInheritedWidgetOfExactType* method belonging to InheritedWidget class.

Source Code 2.46: Todo app - InheritedModel - of method implementation

```
//add the aspect argument
static TodoInheritedData of(BuildContext context, {required int aspect}) {
  final TodoInheritedData? result =
  //calling the inheritFrom method using the aspect parameter
      InheritedModel.inheritFrom<TodoInheritedData>(context, aspect: aspect);
  assert(result != null, 'No TodoInheritedData found in context');
  return result!;
}
```

All the lines of code that access the state with the *of* method must now be changed taking into account the new implementation and the new *aspect* argument.

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

In particular, the TodoView widget will pass as aspect argument the number 0 declaring that should be notified (and rebuild) only when a structural change occurs. Instead, TodoItem widgets will pass the corresponding Todo's *id* in the aspect parameter. Now that every widget is registered to the desired aspect of the data only, is necessary to "teach", the TodoInheritedData widget ,how to recognize aspects changes. To do so, InheritedModel provides a method called *updateShouldNotifyDepenedent* that is just like the InheritedWidget's one, *updateShouldNotify* , but takes as argument also a Set of ints called *dependencies* (aspects). Once a transition occurs this method is called once for

every widget that registered to state changes. The *dependencies* variable will contain all aspects the widget registered to (only one for every widget in our case).

Source Code 2.47: Todo app - InheritedModel - updateShouldNotifyDepented method implementation

```
@override
bool updateShouldNotifyDependent(
    TodoInheritedData oldWidget, Set<int> dependencies) {
  //calculate the length of the current filtered list
  int currLen = filteredTodos.length;
  //calculate the length of the previous filtered list
  int prevLen = oldWidget.filteredTodos.length;
  //if updateShouldNotifyDependent is called in a structural
  // dependent widget and a structural change occured
  bool structureRebuildlen = (dependencies.contains(0) && currLen != prevLen);
  if (structureRebuildlen == true) {
  //rebuild
    return true;
  } else {
    //compute the list of ids in the current filtered list
    List<int> currIds = filteredTodos.map((todo) => todo.id).toList();
    //compute the list of ids in the previous filtered list
    List<int> prevIds =
        oldWidget.filteredTodos.map((todo) => todo.id).toList();
    //compute if ids changed
    bool sameIds = listEquals(currIds, prevIds);
    //computer if updateShouldNotifyDependent is called in a
    // structural dependent widget and a structural change occured
    bool structureRebuildcomp = (dependencies.contains(0) && !sameIds);
    if (structureRebuildcomp == true) {
    //rebuild
      return true;
    } else {
      //create an array of booleans
      List<bool> components = [];
      //for every todo in the current filtered list
```

```
      for (var element in filteredTodos) {
      //add the boolean && between
        components.add(
      //if updateShouldNotifyDependent is called in a widget that registered
      // to the aspect with that id
        dependencies.contains(element.id)
        &&
      // if the element with that id in the previous filtered list is different
            !oldWidget.filteredTodos.contains(element));
      }
      //apply the or operator between all elements in the list
      bool res = components.fold(false,
          (bool previousValue, bool element) => previousValue || element);
      return res;
    }
  }
}
```

This method was tough to code. The method's pseudocode is presented down below.

Source Code 2.48: Todo app - InheritedModel - updateShouldNotifyDepented method pseudocode

```
if( widgetRegisteredForStructuralChange && strucuturalChangeOccured){
        return true;
    }else{
        if( widgetRegisteredForSpecificTodoChange && thatTodoChanged){
            return true;
            }else{
                return false;
    }
}
```

I propose now two pratical examples to clearify a bit the updateShouldNotifyDepented behaviour.

Pratical example 1 - Suppose to have a list of todos containing two todo instances, one with id 38 and one with id 121. In the HomePage there is a TodoView widget containing two TodoItem widgets. TodoView widget calls the *of* method in order to access the state passing the int value 0 as parameter. The two TodoItem widgets call the *of* method in order to access the state passing their todo's id value as parameter. Suppose now to add a new todo in the list using the AddTodoPage. The *updateShouldNotifyDepented* method is called once for every widget that access the state. Note that also the TabSelector and the VisibilityFilterSelector widgets access the state using the *of* method but their are not considered in this example. So, the *updateShouldNotifyDepented* method is called once for the TodoView widget, with dependencies value equal to 0. It is also called once for every TodoItem widget with dependencies value equal to 38, in one case, and equal to 121, in the other. Before actually rebuilding any widget the framework executes all the *updateShouldNotifyDepented* method calls. We start analizing the call for the TodoItem widget with id 38 with reference to the pseudocode RIFERIMENTo. In this case a structural change occured (we added a todo) but the widget did not registered for structural changes (dependencies do not contains 0). Moreover, the widget registered for changes in the todo with id 38 but no changes occured in that todo. The function returns false and the widget is not rebuilt. The same happens for the *updateShouldNotifyDepented* methods execution of the TodoItem widget with id 121. In the *updateShouldNotifyDepented* method execution for the TodoView widget, instead, the dependencies list contains the value 0(and only that). In this case the TodoView widget registered for structural changes and a structural change occured (the new list is longer than the old one). The *updateShouldNotifyDepented* method returns true and the widget is rebuilt creating a new TodoView widget composed by three TodoItem widgets this time.

Pratical example 2 - Suppose to have a list of todos containing two todo instances as before, one with id 38 and one with id 121. In the HomePage there is a TodoView widget containing two TodoItem widgets. TodoView widget calls the *of* method in order to access the state passing the int value 0 as parameter. The two TodoItem widgets call the *of* method in order to access the state passing their corresponding todo's id value as parameter. Suppose now to tap on the TodoItem widget's checkbox with id 38 in order to change its completed value. As before the *updateShouldNotifyDepented* method is called once for every widget that access the state. The *updateShouldNotifyDepented* method execution regarding the TodoView widget returns false because the second if in the Source Code RIFERIMENTo evaluates to false due to the fact that the widget did not registered for specific todo's changes. The TodoView widget is not rebuilt then. In the *updateShouldNotifyDepented* method execution regarding the TodoItem widget with

id 121 the part that leads the entire method to return false is the second expression in the second if statement. Indeed, the todo with id 121 did not changed with respect to the one contained in the previous state. The TodoItem widget with is 121 is consequently not rebuilt. The execution of the *updateShouldNotifyDepented* method concerning the TodoItem with id 38, instead, return true. The second condition is, indeed, satisfied by the fact that the widget registered for a specific todo's change and also that todo changed. The todo with id 38 is, indeed, changed with respect to the todo with id 38 in the previous state. This leads the TodoItem widget with id 38 to rebuild leaving all the other widgets unchanged.

At this point , when the TodoItem's checkbox is tapped the single TodoItem widget is rebuilt. However, no visual changes are shown. The widget rebuilds with the same information as before . This is due to the fact that the *build* method populates its internal widgets based on a local *todo* variable. This variable is populated on the TodoItem widget's creation with a Todo instance provided by the parent widget(TodoView). Indeed, when the TodoView widget instantiates TodoItems in its ListView, it creates a copy of the corresponding Todo before passing it to the constructor. Even if we changed the information contained in the TodoInheritedData widget, the TodoItem widget do not see any difference. Its local todo variable , indeed, did not change. The fact that, before the optimizations, TodoItem widgets rebuilt correctly comes from the fact that the transition of the state caused actions leads the entire TodoView widget to rebuild. The conseguence was that TodoItems were detroyed and created again using new copies of the data coming from the TodoInheritedData widget.
To recap, the performance optimization we were looking for were achieved succesfully but an issue ,regarding the syncronization of the data, arised. The TodoItems widget sees a screenshot of the state different from the one seen by the rest of the application. This is a really bad behavior and is caused by the fact that ,sometimes, during programming , more than one level of information caching is required or used to avoid effort in coding and performance issues. In other words, a local copy of the data is 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 reflect this change. The problem was generate by the fact that a local copy of the real Todo instance was passed to the TodoItem widget. Instead , the "correct" way of handling this scenario is to pass the id of the Todo in the constructor and then use this id to look up for the Todo instance in the centralized state (the TodoInheritedData). This of course will require more computational effort but will guarantee also a lot more stability and robustness. Therefore, TodoItem widget's local variable of type Todo is replaced with a new int variable called *id* that represents the id of the Todo the widget is visualizing. In the *build* method, then, the corresponding Todo is looked up.

Source Code 2.49: Todo app - InheritedModel - TodoItem widget todo look up

```
class TodoItem extends StatelessWidget {
  final int id; // Todo variable replaced with a int one

  const TodoItem({Key? key, required this.id}) : super(key: key);

  @override
  Widget build(BuildContext context) {
          // getting the todo instance from the state
    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 rendering optimizations were successfully accomplished.

Conclusions

Recap

	lines of code	time	classes
base functionalities	86	2-3 h	2
feature addition	43	20-30 m	1
rendering optimization	49	8-10 h	0

Table 2.1: Caption of the Table to appear in the List of Tables.

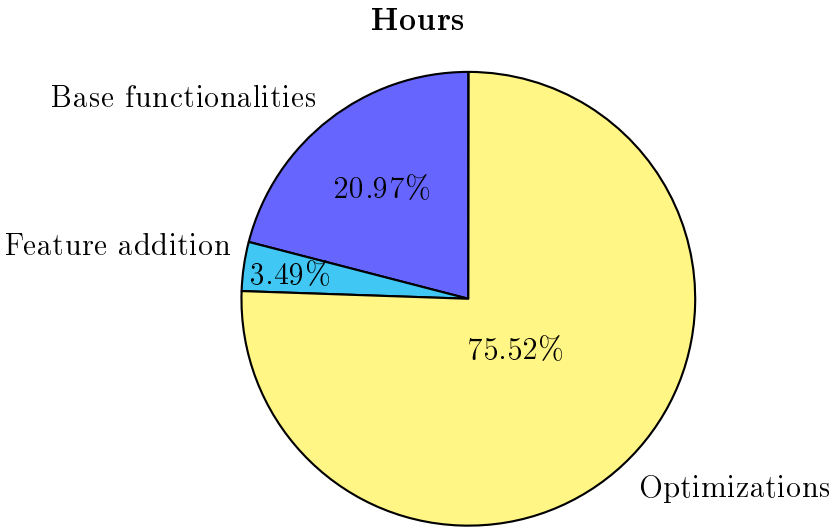


Figure 2.6: Caption of the Table to appear in the List of Tables.

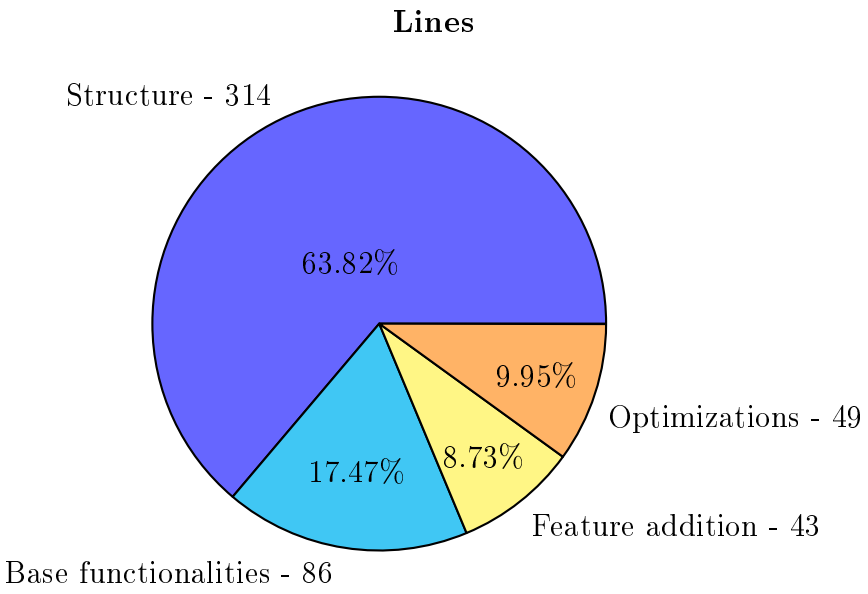


Figure 2.7: Caption of the Table to appear in the List of Tables.

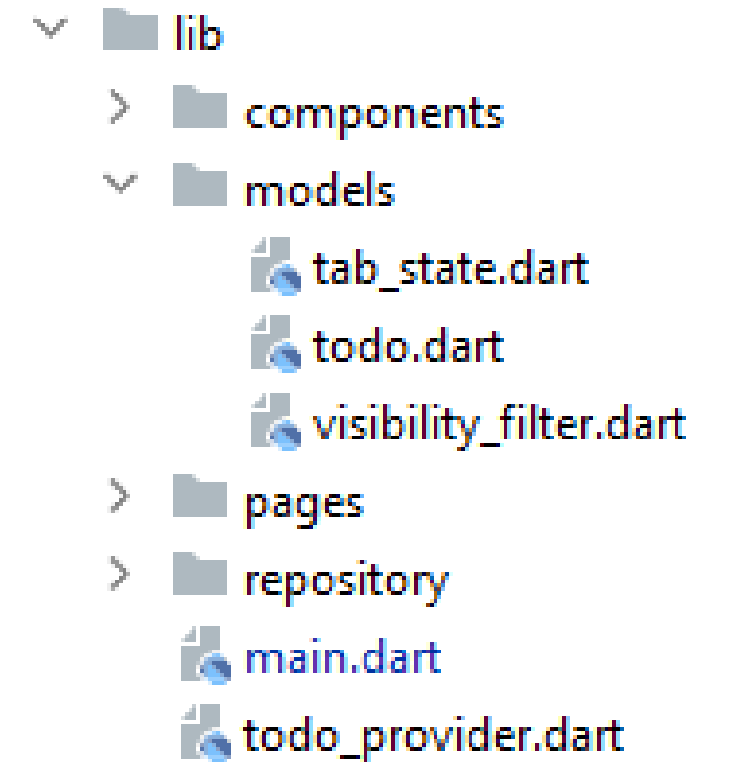
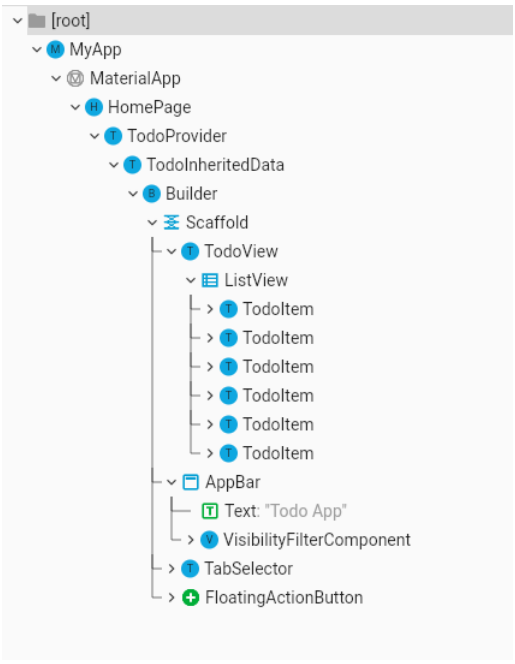
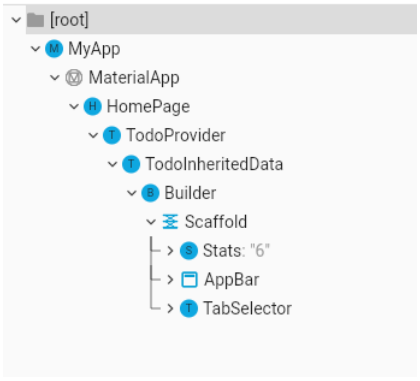


Figure 2.8: Show the tree structure after the FloatingActionButton in the HomePage is tapped.

Down below some images taken from an execution of the application. In this execution ,six todos are randomly created and only two of them are marked as completed.



(a) todos tab runtime UI.



(b) todos tab runtime widget tree.

Figure 2.9: Show the runtime Widget's tree and UI when visualizing todos tab.

Figure 2.21a shows how the application's UI looks like after few seconds from the start. Figure 2.21b show the widget's tree related with the run. Notice the TodoInheritedData widget as a child of the HomePage widget,it provides the state to the subtree.

2.2.3. Redux implementation

In this section the Redux state management solution is used to implement the application.

Base funtionalities

The AppState - Redux solution requires the state to be centrilized in a unique location. A model for this centrilized component must be defined to wrap up all the subparts. 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 groups all the subparts in a unique place throughout the entire application.

Source Code 2.50: Todo app - Redux - AppState model definition

```
class AppState {
```

```
  VisibilityFilter visibilityFilter;
  List<Todo> todos;
  TabState tabState;

  AppState({this.todos=const [],this.tabState= TabState.todos,this.visibilityFilter
}
```

Notice that the list of filtered todos does not take part in the AppState. In Redux, indeed, the centralized state is kept as simple as possible and the parts of the state that can be computed or derived should be omitted. The filtered list of todo should be computed in the presentation layer when needed. This approach, however, can introduce reduntant computations due to the fact that the filtered list is calculated at every widget build. This is where Selectors come into play; they propose a mechanism based on the memoization techinque in order to reuse precomputed value. This aspect will be deeper investigated later.

The actions - In order to mutate the state is necessary to define actions. Actions are processed by reducers to produce new states. Actions are just instances of predefined classes. We start defining the action's classes needed to change the state regarding the list of todos. As usual, the first two features to be implemented are the fetching of todos and the setting of the completed field of a specific todo. Due to the way Redux works, asynchronous actions are handled by Middlewares. Reducers, indeed, are pure functions and are not suited for handling asynchronous code. Two actions are defined just for the fetching process. One is called LoadTodoAction and will be intercepted by the middleware which will take care of fetching the todos from the database. The second is called LoadTodoSuccededAction and carries the list of todos fetched from the database. When the fetching process terminates in the middleware a new LoadTodoSuccededAction is emited and handled by reducers in the AppState.

Source Code 2.51: Todo app - Redux - Loading todos actions definition

```
class LoadTodoAction{
  @override
  String toString(){
    return "loadTodoAction";
  }
```

```
}
class LoadTodoSucceededAction{
  List<Todo> todos;
   LoadTodoSucceededAction(this.todos);
}
```

Another action class is create to handle the setting o the completed field. In this case the action is synchronous and is directly handled by reducers. This new action is called *SetCompletedAction* and contains the id of the todo to be changed and the new value for the completed field.

Source Code 2.52: Todo app - InheritedWidget - SetCompletedTodoAction definitio

```
class SetCompletedTodoAction {
  final int id;
  final bool completed;

  SetCompletedTodoAction(this.id, this.completed);
}
```

Two more action are needed to handle the tab and the visibility filter changes. They are called respectively *SetTabAction* and *SetVisibilityFilterAction*. They contains the new tab value and the new filter value to be set.

Source Code 2.53: Todo app - Redux - SetTabAction definition

```
class SetTabAction{
  final TabState newtab;

  SetTabAction(this.newtab);
}

class SetVisibilityFilterAction{
  VisibilityFilter filter;

  SetVisibilityFilterAction(this.filter);
```

```
}
```

Reducers - Is the turn now for reducer's definition. They will link actions with new states. Even if the usage of the Redux state management solution requires the centralization of the state in a unique component the state's logic can be in any case be split up in subparts. It is like a tree with a single root where the root is represented by the AppState's reducer. It can split up in many sub-reducers and every one of them can further split up into other sub-reducers. Therefore the state is segmented and stored in single place, but its pieces can still be managed separately and independently. During the reasoning process is easier to break the whole state 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, smaller bricks are required 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 reducer 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.

The todoReducer - The reducer for the list of todos needs to handle two actions, the LoadTodoSuccedeedAction and the SetCompletedTodoAction. The TodoReducer is tough a combination of two subreducers which handle a specific action. As we already said, reducers are pure functions. They take the previous state and an action and return the next state. The reducer for the LoadTodoSuccedeedAction is called *setLoadedTodo* and just returns the list contained in the received action.

Source Code 2.54: Todo app - Redux - LoadTodoSuccedeedAction reducer

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

The reducer for the *SetCompletedTodoAction* is called *setCompletedTodo* and takes care of searching in the current todos state the one matching the id contained in the action element. Once the todo is updated the a new instance of the whole list is created and returned. The necessity to create another instance comes from the fact that, in case the state's list is just updated and not substituted, the transition would not be recognised.

Source Code 2.55: Todo app - Redux - SetCompletedTodoAction reducer

```
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);
}
```

Now that all the needed reducers has been implemented we use some tools, the Redux package provides, in order to combine them in a single 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 avoiding nested if-else structures 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 new reducer composing sub-reducers provided in the form of *TypedReducers*. The two previously defined reducers are now merged into a single one called *todosReducer*.

Source Code 2.56: Todo app - Redux - combine reducers using combineReducers function

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

the alternative would have been to write the *todosReducer* as presented RIERIEMENTO. The output is the same but the code is clearly less readable.

Source Code 2.57: Todo app - Redux - combine reducers using the traditional way

```
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;
  }
};
```

The tab Reducer and the visibility Filter reducer - The process is the same as before. Two reducers, called *setTabState* and *setVisibilityFilter*, are created to handle actions of type SetTabAction and SetVisibilityFilter respectively. 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.

Source Code 2.58: Todo app - Redux - reducers definition for the tab and the filter state

```
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)
]);
```

The AppState reducer - All the basic bricks have been implemented and can be used now to create the biggest reducer; the AppState reducer. It is a function that takes the current *AppState* and an action. It then composes a new AppState instance using the sub-reducers it is composed of. Every sub-reducer processes the received action in order to understand if it needs to perfom a transition in the part of the state it handles.

Source Code 2.59: Todo app - Redux - AppState definition

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

The Middleware - All the necessary to start accessing the state in the presentation layer is settled up but the way in which the todos are fetched from the database is not clear yet. Two actions were set up for this purpose but only one has been handled by a reducer. 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 asynchronous code using Redux but, the most correct one (and also the one proposed in the documentation) is to use Middlewares. Middlewares has been introduce HERE RIEFERIMENTO. They act as a sort of proxy between actions dispatch and reducers. One or more middlewares can be set to execute on actions call. They are used to handle asynchronous code but also action that generate side effects. In our case, an action of type LoadTodoAction has been defined but is not handled by any reducer yet and is ignored once received. However, if dispatched, it will pass through one or more middleware before reaching the reducers. Is necessary tough to set up a middleware that intercepts it and starts the fetching process. 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 then pass the action to the next middleware. A new middleware function is defined and called *loadTodoMiddleware*. It checks if 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 an action of type LoadTodoSuccededAction that contains the list of fetched todo. We previously set up a specific reducer in order to handle this type of action in the AppState reducer.

Source Code 2.60: Todo app - Redux - loadTodosMiddleware middleware definition

```
void loadTodosMiddleware(Store<AppState> store, action, NextDispatcher next) {

  if (action is LoadTodoAction) {
    TodoRepository.loadTodos().then((todos) {store.dispatch(LoadTodoSucceededAction
  }
  next(action);
}
```

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

Making the state accessible - Redux uses Provider widgets to make the state accessible down the widgets tree. The state is unique and tough should 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 widget is positioned at the root of app, right below the MyApp widget. An object of type Store must be provided to the StoreProvider widget, in the *store* field. The typed class Store is provided by the redux package. 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 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 at the application start we will use the *init* method of stateful widgets. The HomePage is already stateful and its *init* method is used in order 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.

Source Code 2.61: Todo a pp - Redux - Widgets tree root definition

```
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());
            },),
            (...)
         },
      ),
    );

  }
}
```

Inside the HomePage, the *initState* method is set up in order to dispatch a LoadTodoAction using the function passed as parameter by the MaterialApp widget. In order to access the store, a StoreConnector widget is used. A StoreConnector widget takes two functions in its *converter* and *builder* fields. The *converter* function takes the state and creates a viewmodel containing the minimal information necessary to build the widget. The *builder* function actually uses the viewmodel in order to create the widget.

The viewmodel is really important because idenfies the prospective from which, the widget, is looking at the centralized state. If the viewmodel changes the widget is rebuilt. Not always a state change produces a viewmodel change. The StoreConnector widget is a typed widget also and takes two types in its definition/usage: the store's type, the AppState, and the viewmodel's type, in our case just a TabState object. The HomePage, indeed, only needs the part of the state related to the tab and, for this reason, it is not necessary to create an ad hoc viewmodel. The model of the TabState defined hereRIFERIMENTO is enough to contain the interesting part of the state. In the *converter* field a function is provided which manipulates the state returning the current TabState value. The *builder* field is populated with a function that returns a Scaffold widget. Inside this function, the current context and the object returned from the converter function can be used. This last one is used to populate the Scaffold widget as usual.

Source Code 2.62: Todo app - Redux - state integration in the HomePage

```
@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(...),
          body: currTab == TabState.todos ? const TodoView() : const Stats(),
          bottomNavigationBar: const TabSelector(),
          floatingActionButton: currTab == TabState.todos
              ? FloatingActionButton(...)
              : Container());
    },
  );
```

```
}
```

We can now start implementing the component widgets using the AppState but, first, a short digression about Selectors is taken.

Selectors - They are introduced hereRIFERIMENTO . Selectors are used to compute those parts of the state that entirely depends on other parts. In our case, for example, the filtered list is not included in the AppState with the idea of computing it in the presentation layer. This decision also comes from the fact that would be hard to syncornize it with the changes in the todos list and in the filter if situated in the centralized state. The easiest way to deal with the filtered list is to compute it in the UI layer before building 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 widget is rebuilt. In this scenario Selectors come into play to memoize the previously computed value and to reuse them once accessed. Selectors are just functions that take as input the state an return an element composed with it. All the memorization part is perfomed by a third party package called *Reselect*. In order to use this package it must be included in the pubspec.yaml file under the *dependencies* field. It provides some functions with the called *createSelector* that take care of memoizing some values and understanding when to recompute them. Selectors can be simple o composed. For example, two simple selectors can be implemented in our case. One takes the state and return the filter and the other takes the state and returns the list of todos.

Source Code 2.63: Todo app - Redux - base Selectors definition

```
final todosSelector = (AppState state) => state.todos;

final filterSelector = (AppState state) => state.visibilityFilter;
```

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 *todosSelector* and the *createSelector1* function. The same con be done to compute the list of pending todos. Another Selector is created to finally compute the filtered list. It uses the four other selectors we just implemented and the *createSelector4* function. Besides of making the code more readable , selectors, allow to optimize the application's performances.

Source Code 2.64: Todo app - Redux - composed Selectors definition

```
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;
  }
});
```

The TodoView component - After this short digression about selectors, we are ready to set up the TodoView component. The ListView widget is wrapped into a StoreConnector widget typed with the AppState and List<Todo> types. The TodoView component just needs to access the part of the state concerning the filtered list of todos that is representable with an object of type List<Todo>. The list of filtered todos is not directly available in the AppState but selectors can be used to compute it. In the

converter field of the StoreConnector widget, a function is provided which takes the store and returns the filtered list using the *filteredTodosSelector*. The *converter* function output is available inside the *builder* field's function and is used to populate the ListView widget as usual.

Source Code 2.65: Todo app - Redux - state integration in the TodoView component

```
class TodoView extends StatelessWidget {
  const TodoView({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return StoreConnector<AppState, List<Todo> >(
        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);
        });
  }
}
```

The TodoItem component - The TodoItem component does not require any modification with respect to the implementation proposed HERE RIFERIMENTO. It just receives a todo from the parent widget and exposes it to the user. The only missing part is the *onChanged* field of the Checkbox widget which is still to be implemented. Inside the *onChage* field, a function must be provided which updates the *completed* field of the visualized todo. The store is accessed using the *of* method of the StoreProvider widget

(the of method gets the nearest instance of provided type, 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.

Source Code 2.66: Todo app - Redux - state integration in the TodoItem component

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

The VisibilityFilterSelector component - The VisibilityFilterSelector 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* field is filled with a function taking the store and returning the current filter value. To do so, it uses the *filterSelector* implemented earlier. The output of the *converter* function is accessed in the *builder* field function and used to populate the DropdownButton widget. The function used in the *onChanged* field of the DropdownMenuItem widgets gets an instance of the store using the StoreProvider widget's *of* method and dispatches an action of type SetVisibilityFilterAction using the DropdownMenuItem 's filter value.

Source Code 2.67: Todo app - Redux - state integration in the VisibilityFilterSelector component

```
@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: (...),
        onChanged: (filter) {
          StoreProvider.of<AppState>(context)
              .dispatch(SetVisibilityFilterAction(filter!));
        },
      );
    },
  );
```

The TabSelector component - Also in this case, the TabSelector component requires to read and write the AppState. A StoreConnector widget is used to wrap the BottomNavigationBar widget and to connect it to the state. The viewmodel to be returned by the *converter* function is just an object of type TabState. An arrow function returning the current AppState's tab value is provided to the converter field. The function output is then used in the *builder* field's function to populate the BottomNavigationBar widget. The *onTap* field of the BottomNavigatioBar widget is filled with a function that dispatches an action of type SetTabAction when fired.

Source Code 2.68: Todo app - Redux - state integration into TabSelector component

```
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(TabS
          items: (...)
```

```
        );
      },
    );
  }
}
```

The Stats component - The stats component only requires to read the state. The Center widget is wrapped into a StoreConnector widget. In the *converter* field's function the *completedTodoSelector* selector is used to access the list of completed todos. In this case the viewmodel to be outputted is just an int value representing the number of completed todos.

Source Code 2.69: Todo app - Redux - state integration into Stats component

```
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;
      },
    );
  }
}
```

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

Features addition

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

Source Code 2.70: Todo app - Redux - AddTodoAction and UpdateTodoAction definition

```
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);
}
```

In order to handle these new actions two new reducers must be created. They are called respectively *addTodo* and *updateTodo*. The *addTodo* reducer first creates a new todo using the information contained in the *AddTodoAction* instace and a newly generated unique id. Then, it creates a new list instace and populates it with the elements of the old list plus the new todo.

Source Code 2.71: Todo app - Redux - AddTodoAction's reducer definition

```
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);
}
```

The *updateTodo* reducer first modifies the todo with the id matching the one contained in the action. Then, it generates a new list and fills it with the element contained in the current one.

Source Code 2.72: Todo app - Redux - UpdateTodoAction's reducer definition

```
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);
}
```

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

Source Code 2.73: Todo app - Redux - combining new reducers to the old ones

```
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),
]);
```

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. The StoreProvider widget has been positioned in the root of the application to be available in all the subtrees. Is sufficient, tough, to access the store in the AddTodoPage and dispatch an action of type AddTodoAction, when the TextButton widget is tapped.

Source Code 2.74: Todo app - Redux - using the AddTodoAction in the AddTodoPage

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

The same is done in the UpdateTodoPage. Once the TextButton widget is tapped an action of type UpdateTodoAction is dispatched.

Source Code 2.75: Todo app - Redux - using the UpdateTodoAction into the UpdateTodoPage

```
TextButton(
    onPressed: () {
      final UpdateTodoAction action= UpdateTodoAction(textControllerName.text,textContro
      StoreProvider.of<AppState>(context).dispatch(action);
```

```
      Navigator.pop(context);
    },
    child: const Text("Confirm"))
```

All the features have been successfully added.

Rendering optimization

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 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 provide it with the correct equality operator. Some changes must be done in the TodoView widget and in the TodoItem widget. Firstly, the TodoItem widget needs to interact with the state in order 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 exposes it to the user.A StoreConnector widget is used to read the state in the TodoItem widget. The complete instance of the todo to be visualized is not necessary 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.

Source Code 2.76: Todo app - Redux - make the TodoItem component read the state

```
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(. . .);
        });
  }
}
```

Now that the TodoItem widget listens for its own todo changes, the optimization process concerning 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, once a new state is emitted, the list of todos is recreated from scratch and its internal todos are replaced with new instances. 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 autonomously. The same process must be done in the TodoView widget. It already uses a StoreConnector widget to derive the list of filtered todos from the store. However, as we just said, two different plain lists appear to be always different by default also if their internal aspects matches. Moreover, it is not possible to redefine the equality operator for lists and, in case, it would not be a good idea. A simple way to handle this scenario is to create an ad hoc viewmodel and redefine its equality operator in order to match our own rebuilding logic. To do so, a local class is created and called ViewModel. This local class just contains a list of todos.

Source Code 2.77: Todo app - Redux - TodoView ad hoc ViewModel definition

```
class _ViewModel {
```

```
  final List<Todo> todos;

  _ViewModel({required this.todos});

}
```

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).

Source Code 2.78: Todo app - Redux - ViewModel equality operator override

```
  @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;
```

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.

Source Code 2.79: Todo app - Redux - TodoView renders optimization

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

To put the the icing on the cake we set the *distinct* field to true also in the VisibilityFilterSelector component and in the TabSelector component. In this way they are rebuilt only in case their prospective of the state changes and not at every state transition.

Conclusions

Recap

	lines of code	time	classes
base functionalities	156	9-11 h	6
feature addition	50	15-20 m	2
rendering optimization	28	1 h	1

Table 2.2: Caption of the Table to appear in the List of Tables.

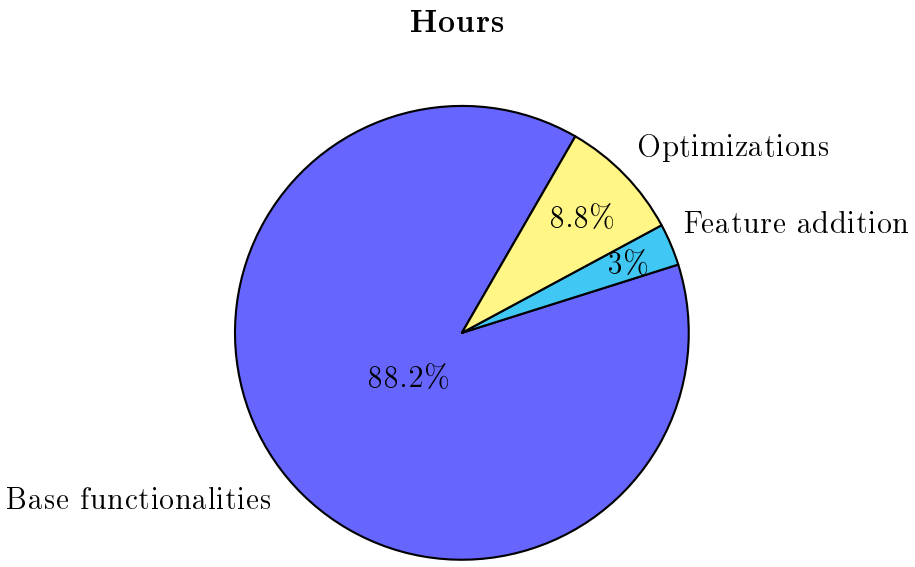


Figure 2.10: Caption of the Table to appear in the List of Tables.

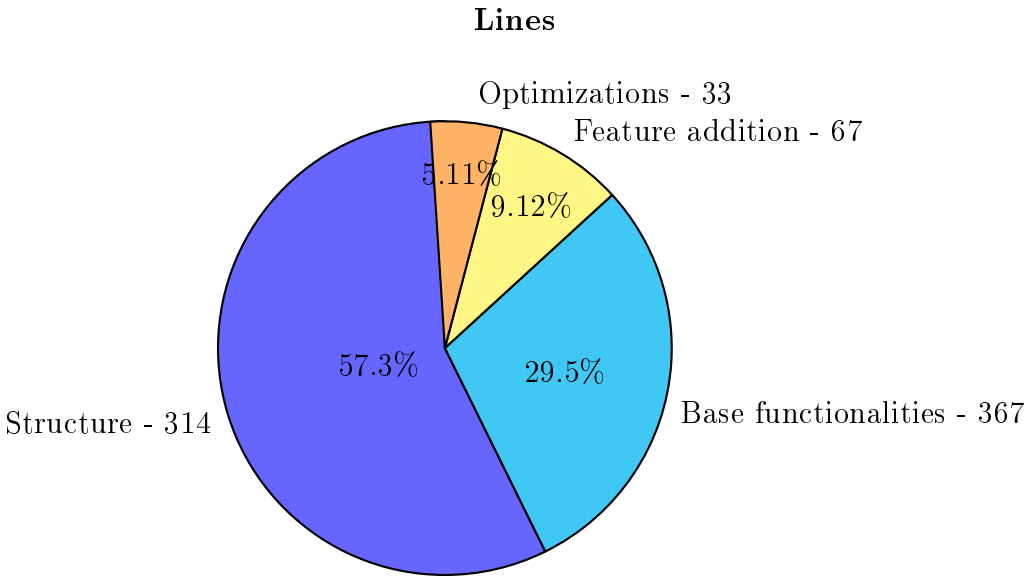


Figure 2.11: Caption of the Table to appear in the List of Tables.

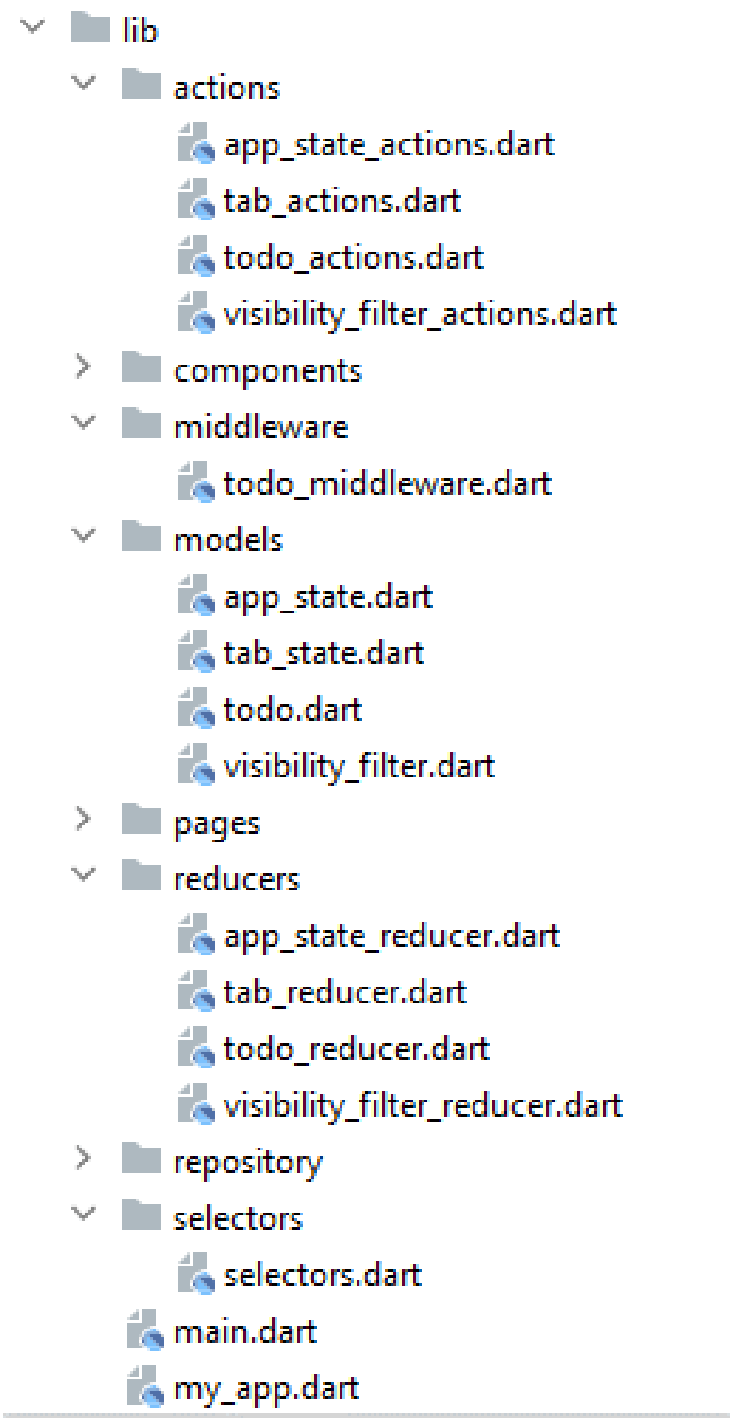
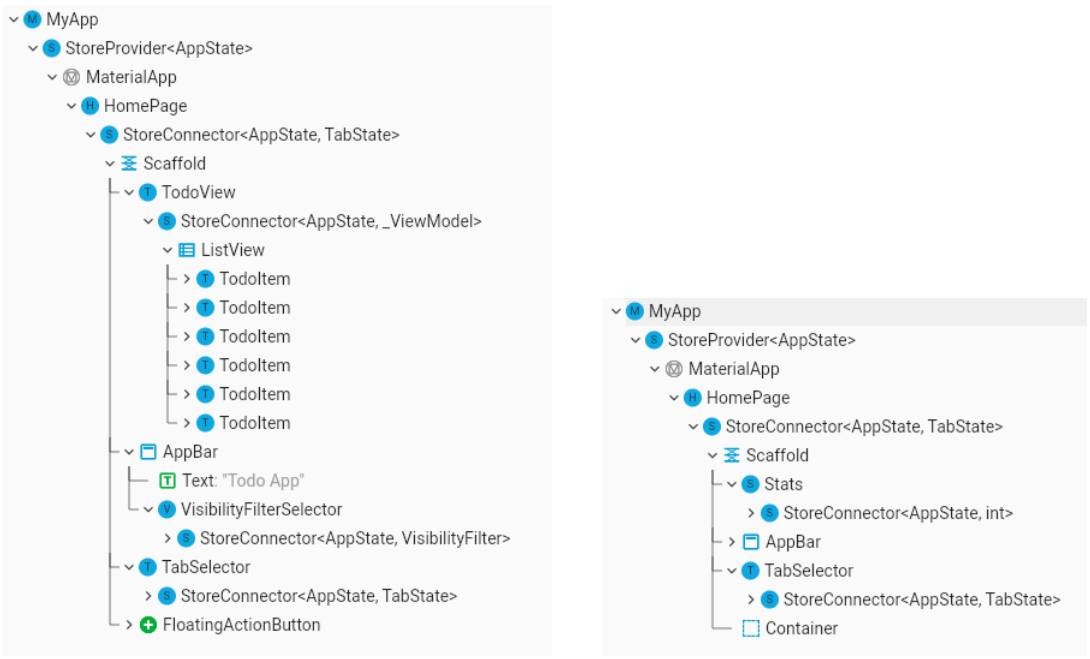


Figure 2.12: Show the tree structure after the FloatingActionButton in the HomePage is tapped.

Down below some images taken from an execution of the application. In this execution ,six todos are randomly created and only two of them are marked as completed.



(a) todos tab runtime UI. (b) todos tab runtime widget tree.

Figure 2.13: Show the runtime Widget's tree and UI when visualizing todos tab.

Figure 2.21a shows how the application's UI looks like after few seconds from the start. Figure 2.21b show the widget's tree related with the run. Notice the TodoInheritedData widget as a child of the HomePage widget,it provides the state to the subtree.

2.2.4. BloC implementation

In this section the state management solution called Bloc is used to implemented the todo application. The bloc overview can be found here RIFERIMENTO.

Base funtionalities

States - 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 statistics 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 of 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.

The states of the list of todos - First of all we start defining and naming the possible states of the list of todos. These states 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 states a new abstract class is created. It is called TodosState. It must extend the class Equatable. The Equatable class is useful to define equality between states without the need to override the equality operator in every state class. The TodosLoadingState does not contains any other information. The TodosLoadedState contains ,instead, a list filled with todos.

Source Code 2.80: Todo app – Bloc - states definition for the list of todos

```
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';
}
```

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 the others state classes ,belonging to the state relative to the filtered list and the filter, will extend the FilteredTodosState abstract class. Someone can notice that , the state of the filtered list and the filter ,contains two different aspects of the application state: the filter and the filtered list precisely. 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 the more 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 , indeed, 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 also, making its dependencies going from one bloc to two blocs (the bloc of the todos and the bloc of the filter).

Source Code 2.81: Todo app - Bloc - states definition for the filtered list of todos and 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 . The second identifies the fact that stats are available and contains an int variable inside , *completed* that represents the actual stats.

Source Code 2.82: Todo app - Bloc - states definition for the stats

```
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 ,the enumeration presented HERE RIFERIMENTO is enough.

Source Code 2.83: Todo app - Bloc - state definition for the tab

```
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 can represent a specific actions the user can perform or also internal changes. They enable the states to mutate creating transitions.

Events of the list of todos - For the moment is sufficient to define two events 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 to be modified 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 abstract class.

Source Code 2.84: Todo app - Bloc - event definition for the list of todos

```
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. A new filtered list must be computed ,tough, and a new FilteredTodosLoadedState emitted. It contains internally a variable providing the changed list of todos.
Also in this case ,all event classes extend a shared abstract class called FilteredTodoEvent which , in turn, extends the Equatable class.

Source Code 2.85: Todo app - Bloc - events definition for the filtered list of todos and the filter

```
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:
}

class TodoUpdatedEvent extends FilteredTodoEvent {
  final List<Todo> todos;

  @override
  List<Object> get props => [todos];

  const TodoUpdatedEvent(this.todos);

  @override
  String toString() => 'FilteredTodoEvent - TodoUpdatedEvent';
}
```

Events for the stat'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 in the TodoBloc. Precisely it is generated once a new state of type TodosLoadedState is emitted in the TodoBloc. It contains internally the new list of todos of the emitted state.
Also in this case , both the event for the stats and the event for the tab extends respectively the abstact classes TabEvent and StatsEvent.

Source Code 2.86: Todo app - Bloc - events definition for the stats and the tab

```
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 the way in which 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, 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 where 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 *mapEventToState* method is ,indeed, annoted with the @override notation meaning that the implementation we are giving substitutes the one of the Bloc class. 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* statement 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 terminates and the application become unresponsive. For code readability, the logic to be executed when a 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 blocs 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 it. In case of failure ,instead, the TodosLoadingState is emitted. The *mapSetCompletedToState* method takes as single argument an event 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 value 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 list present in the previous state the Equatable class do not identify any difference between the previous state and the new emitted one, and consequently, do not notify any listener.

Source Code 2.87: Todo app - Bloc - TodoBloc implementation

```
class TodoBloc extends Bloc<TodosEvent, TodosState> {
  //initialize the bloc's state at creation
  TodoBloc() : super(TodosLoadingState());

  @override
  Stream<TodosState> mapEventToState(TodosEvent event) async* {
    if (event is LoadTodosEvent) { // if an event of type LoadTodosEvent is receiver
      yield* _mapLoadTodosToState(event);
    } else if (event is SetCompletedTodoEvent) {/ if an event of type SetCompletedTodoEv
      yield* _mapSetCompletedToState(event);
    }
  }

  Stream<TodosState> _mapLoadTodosToState(LoadTodosEvent event) async* {
    try {
    //fetch todos
      final List<Todo> todos = await TodoRepository.loadTodos();
      yield TodosLoadedState(todos);
    } catch (e) {
```

```
      yield TodoLoadingState();
    }
  }

Stream<TodosState> _mapSetCompletedToState(
    SetCompletedTodoEvent event) async* {
  if (state is TodosLoadedState) {// create a new updated list
    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 at the initialization and used 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 with respect to the TodoBLoc's one. It emits the initial state based on the state of TodoBloc. If the state is of type loaded , the constructor computes ,and then emits , a state of type FilteredtodosLoadedState using a filter of type *all*. If the state is of type loading, the constructor emits a state of type FilteredLoadingState.

Source Code 2.88: Todo app - Bloc - FilteredTodoBloc initial state definition

```
class FilteredTodoBloc extends Bloc<FilteredTodoEvent, FilteredTodoState> {
```

```
  //this variable is used to listen for changes in the todoBloc
  final TodoBloc todoBloc;

  FilteredTodoBloc({required this.todoBloc})
      : super(
      //initialize the state based on the todoBloc's state
          todoBloc.state is TodosLoadedState
              ? FilteredTodoLoadedState(
                  (todoBloc.state as TodosLoadedState).todos,
                  VisibilityFilter.all,
                )
              : FilteredTodoLoadingState(),
        )
}
```

In addition, the constructor must register the bloc to the changes in the TodoBLoc. To do so , a particular variable of the TodoBloc instance , called *stream*, is used. The variable *stream* is present because the TodoBloc extends the Bloc class. It is , indeed, the 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 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 ,instead, we emit a new event that will be handled by the *mapEventToState* method ,defined later. Once a new state is emitted in the TodoBloc the function checks 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 updated. In both cases , the bloc of the filtered list must compute a new filtered list and emit a new state containing it. A specific event ,called TodoUpdatedEvent ,has been defined for this situation HERE RIFERIMENTo where the events related to the bloc of the filtered list have been implemented.

Source Code 2.89: Todo app - Bloc - FilteredTodoBloc subscription to TodoBloc stream

```
class FilteredTodoBloc extends Bloc<FilteredTodoEvent, FilteredTodoState> {
  final TodoBloc todoBloc;
  //is marked as late because is initialized after the constructor execution
  late StreamSubscription todoSubscription;
```

```
  FilteredTodoBloc({required this.todoBloc})
      : super(
          todoBloc.state is TodosLoadedState
              ? FilteredTodoLoadedState(
                  (todoBloc.state as TodosLoadedState).todos,
                  VisibilityFilter.all,
                )
              : FilteredTodoLoadingState(),
        ) {
        //subscription to the todoBloc's state changes
    todoSubscription = todoBloc.stream.listen((state) {
    //in case a new state of type Loaded is emitted
      if (state is TodosLoadedState) {
      //an internal event of type TodoUpdatedEvent is emitted
        add(TodoUpdatedEvent((todoBloc.state as TodosLoadedState).todos));
      }
    });
  }
```

The *mapEventToState* method is overriden now 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 returns a stream of FilteredTodosStates. The method takes as argument a single event of the generic abstract type FilteredTodosEvent. Inside the method, two nested *if-else* statement defines the type of the received event. 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.

Source Code 2.90: Todo app - Bloc - FilteredTodoBloc *mapEventToState* method implementatio

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

```
    yield* _mapTodoUpdatedEventToState(event);
  }
}
```

The *mapTodoChangeFilterEventToState* method checks that the state of the TodoBloc 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 todos.

Source Code 2.91: Todo app - Bloc - FilteredTodoBloc _ *mapTodoChangeFilterEvent-ToState*method implementation

```
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 filter of type *all*.

Source Code 2.92: Todo app - Bloc - FilteredTodoBloc _ *mapTodoUpdatedEventToState* method implementatio

```
Stream<FilteredTodoState> _mapTodoUpdatedEventToState(
    TodoUpdatedEvent event) async* {
    // populate a filter value bases on the todoBloc's state
  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.

Source Code 2.93: Todo app - Bloc - FilteredTodoBloc *close* method implementation

```
@override
Future<void> close() {
// cancel the subscription
  todoSubscription.cancel();
  return super.close();
}
```

The bloc for the stats - This bloc is similar to the previous one , it has to deal with one event only, tough : the StatsUpdatedEvent. As usual, the class StatsBloc is defined an extended with the Bloc class. The StatsBloc class handles events 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 required in the constructor. In the constructor, a new initial state of type StatsLoadeingState is emitted. The subscription to the state's 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 it is ,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 of type TodosLoadedState on StatsBloc's creation.

Source Code 2.94: Todo app - Bloc - StatsBloc constructor implementation

```
class StatsBloc extends Bloc<StatsEvent, StatsState> {
  final TodoBloc todoBloc;
```

```
  //marked as late because it initialized after constructor exectuion
  late StreamSubscription todoSubscription;

  StatsBloc({required this.todoBloc}) : super(StatsLoadingState()) {
  // wrap the code into a function to reuse it
    void onTodosStateChanged(state) {
      if (state is TodosLoadedState) {
        add(StatsUpdatedEvent(state.todos));
      }
    }

    onTodosStateChanged(todoBloc.state);
         // is a new event is emitted in the todoBloc execute the onTodosStateChanged fu
    todoSubscription = todoBloc.stream.listen(onTodosStateChanged);
  }
```

The *mapEventToState* method requires a single *if-else* statement because the only event it has to handle is the StatsUpdatedEvent. When received, the list of todos it contains is used to compute the stats and ,then, a new StatsLoadedState is emitted. In the *close* method the subscription to the TodoBloc is terminated.

Source Code 2.95: Todo app - Bloc - StatsBloc *matEventToState* and *close* methods implementation

```
@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() {
  //cancel the subscription
```

```
    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 to be handled are specified. In the contructor, the initial state is initialized and set to the TabState.todos value. The *mapEventToState* method is overridden connecting the only event with the emission of a state corresponding to the event's TabState internal value.

Source Code 2.96: Todo app - Bloc - TabBloc implementation

```
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 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.

Source Code 2.97: Todo app - Bloc - example of TodoBloc usage

```
void main()  {
  //create a new bloc
  TodoBloc todoBloc= TodoBloc();
```

```
  //dispatch an event
  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 the bloc package is. It 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, it 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 intrested bloc. *onTransition* is called everytime a state transition occurs, inside a bloc. It 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 one event. The states are the ones preceeding and postponing the the event's 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 occured and an object of type StackTrace that reports the stack situation when the error occurred. In our case the corresponding event, transition and error are displayed only but other, more articolated , implementation can be provided.

Source Code 2.98: Todo app - Bloc - AppBlocObserver implementation

```
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.

Source Code 2.99: Todo app - Bloc - application's observer setting

```
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 situated in the root. In the *main* function , the first widget to be passed to the *runApp* method is indeed a BlocProvider widget. A BlocProvider widget is a typed widget , meaning that, the type of the bloc it makes accessible ,must be provided. In our case it needs to provide a bloc of type TodoBloc. Inside the BlocProvider widget , two fields must be filled: *create* and *child*. In the *create* field, a function taking as single argument the context and returning a bloc of the previously 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 widget 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. It is possible to set the lazy flag to false in case of the *create* function needs to be run instantly after the

widget build. A function that instantiates a TodoBloc and emits the first event of the application, the LoadTodoEvent, is provided in the *create* field. 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 using the ".." notation. The *child* field is populated with the MyApp widget as usual.

Source Code 2.100: Todo app - Bloc - make the TodoBloc accessible

```
void main() async {
//set the  observer
  Bloc.observer = AppBlocObserver();
  //run di application and fetch todos
  runApp( BlocProvider<TodoBloc>(create:(context)=>
   TodoBloc()..add(LoadTodosEvent()),child: const MyApp()));
}
```

Beyond the TodoBloc, also the other blocs previously defined need to be made accessible. They are required in the HomePage only because the information they provide is not used by the other pages. This time a MultiBlocProvider 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 is inserted. It is the same as nesting a series of BlocProviders widge but it has the advantage of making the code more readable. In the *providers* field, a list with three BlocProvider widgets is inserted. The first is of type TabBloc, the second is of type StatsBloc and the third is of type FilteredTodoBloc. The last two BlocProvider widgets need to 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 for 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. 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 allows the state to be modified by the parts that has access to it only and , in case of problems, it is easier to understand which part of the code caused it.

Source Code 2.101: Todo app - Bloc - make other blocs accessible

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("building: MATERIAL-APP");
    return MaterialApp(
      initialRoute: "/",
      routes: {
      // usage of MultiBlocProvider
        "/": (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)),
      },
    );
  }
}
```

The state intergration inside the UI - Now that the application's state has been defined and also made accessible in the interested part of the widgets trees 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 build on top of the tab value. 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 its declaration. Inside the function provided in 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.

Source Code 2.102: Todo app - Bloc - wrapping the HomePage into a BlocBuilder

```
class HomePage extends StatelessWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print("building: HomePage");

    return BlocBuilder<TabBloc, TabState>( //use a BlocBuilder
    builder: (context, tabState) {
            return Scaffold(...);
          });
  }
}
```

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

Source Code 2.103: Todo app - Bloc - HomePage's Scaffold based on the tab

```
builder: (context, tabState) {
//using the tabState
  return Scaffold(
    appBar: AppBar(
      title: const Text("TodoApp"),
      actions:  [tabState == TabState.todos?  //here VisibilityFilterComponent():Contair
    ),
    //here
    body: tabState == TabState.todos ? const TodoView() : const Stats(),
    bottomNavigationBar: const TabSelector(),
    floatingActionButton: //here
```

```
        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 ,using 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 an *if-else* 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 inside the filteredTodosState object and can be used to populated the ListView widget.

Source Code 2.104: Todo app - Bloc - TodoView implementation

```
class TodoView extends StatelessWidget {
  const TodoView({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocBuilder<FilteredTodoBloc, FilteredTodoState>(//using a BlocBuilder
        builder: (context, filteredTodoState) {
      print("building: TodoView");
        //dependeing on the current state
      if (filteredTodoState is FilteredTodoLoadedState) {
        return ListView.builder(
            itemCount: filteredTodoState.todos.length,//access it here
```

```
            itemBuilder: (context, index) {
              return TodoItem(
                  todo: filteredTodoState.todos.elementAt(index)); //here
            });
      } else if (filteredTodoState is FilteredTodoLoadingState) {
        return const Center(child: CircularProgressIndicator());
      } else {
        return const Center(child: CircularProgressIndicator());
      }
    });
  }
}
```

The TodoItem component - Since this part of the development process does 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. 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 passed by the parent ,and emitted in the TodoBloc.

Source Code 2.105: Todo app - Bloc - onChanged field implementation inside a TodoItem's Checkbox

```
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 rebuilt showing the modification.

The VisibilityFilterSelector component - The VisiblityFilterSelector component 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 in case a DropdownMenuItem is tapped. The DropdownButton is wrapped inside a BlocBuilder widget. The BlocBuilder widget is informed ,with the <> notation , it will handle the bloc of type FilteredTodosBloc and the states of type FilteredTodoState.

Source Code 2.106: Todo app - Bloc - wrapping the VisibilityFilterSelector component into a BlocBuilder

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

Inside the *builder* field, 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*.

Source Code 2.107: Todo app - Bloc - populating a filter variable based of the current state

```
final VisibilityFilter filter= filteredTodoState is FilteredTodoLoadedState? filter
```

The DropdownButton is populated with the created filter variable. Notice that ,the function provided in the *onChenge* field of every DropdownMenuItem widget, uses its

internal filter value to create ,and emit, a new event in the FilteredTodoBloc of the type FilteredTodoChangeFilterEvent.

Source Code 2.108: Todo app - Bloc - DropdownMenuItem's onChange field implementation

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

The TabSelector component - The entire component depends only on the state of the tab. It needs to read and write the state. The BottomNavigatorBar widget is wrapped inside a BlocBuilder widget. the BlocBuilder widget will handle the bloc of type TabBloc and the states of type TabState.

Source Code 2.109: Todo app - Bloc - wrapping the BottomNavigationBar into a BlocBuilder

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

TheBottomNavigationBar's *onTap* field is populated with a function that emits a new event of the type ChangeTabEvent ,inside the TabBloc, after the user taps the BottomNavigationBarItem.

Source Code 2.110: Todo app - Bloc - BottomNavigationBar's *onTap* field implementation

```
onTap: (index)=>BlocProvider.of<TabBloc>(context).add(ChangeTabEvent(TabState.values.ele
```

The Stats component - Also in this case, the only dependency the Stats component has is about the part of the state concerning the stats. The component is, therefore,

wrapped into a BlocBuilder widget. The Blocbuilder widget will handle the bloc of type StatsBloc and the states of type StatsState. Inside the function provided into 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 and populated using the *completed* variable contained inside the state object. In case the state is of type StatsLoadingState a CircularProgressIndicator widget is returned ,indicating that the stats still need to be computed.

Source Code 2.111: Todo app – Bloc - Stats component implementation

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

Features addition

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

New events - 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.

Source Code 2.112: Todo app - Bloc - AddTodoEvent and UpdateTodoEvent definition

```
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';
}
```

Bloc update - 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.

Source Code 2.113: Todo app - Bloc - TodoBloc mapEventToState new feature extension

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

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.

Source Code 2.114: Todo app - Bloc - *mapTodoAddedToState* method definition into the todoBloc

```
Stream<TodosState> _mapTodoAddedToState(AddTodoEvent event) async* {
    if (state is TodosLoadedState) {
    //generate new unique id
      int newId = generateId((state as TodosLoadedState).todos);
      //create new todo
      Todo newTodo = Todo(
          id: newId,
```

```
          name: event.name,
          description: event.desc + " " + newId.toString(),
          completed: false);
          //add it to a new list instance
      final List<Todo> updatedTodos =
          List.from((state as TodosLoadedState).todos)..add(newTodo);
      yield TodosLoadedState(updatedTodos);
    }
  }
```

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.

Source Code 2.115: Todo app - Bloc - *mapTodoUpdatedToState* method definition into the todoBloc

```
Stream<TodosState> _mapTodoUpdatedToState(UpdateTodoEvent event) async* {
  if (state is TodosLoadedState) {
  //replace the list with a new updated one
    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);
  }
}
```

Access new feature in the UI - 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.

Source Code 2.116: Todo app - Bloc - emitting an AddTodoEvent in the AddTodoPage

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

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.

Source Code 2.117: Todo app - Bloc - emitting an UpdateTodoEvent in the UpdateTodoPage

```
TextButton(
    onPressed: () {
      BlocProvider.of<TodoBloc>(context).add(UpdateTodoEvent(
```

```
          widget.todo.id,
          textControllerName.text,
          textControllerDesc.text));
      Navigator.pop(context);
    },
    child: const Text("Confirm"))
```

Rendering optimization

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.

Source Code 2.118: Todo app - Bloc - Making the TodoItem widget access the state

```
class TodoItem extends StatelessWidget {
  final int id; //todo variable replaced with int one
```

```
  const TodoItem({Key? key, required this.id}) : super(key: key);

@override
Widget build(BuildContext context) {
//new BlocBuilder widget
  return BlocBuilder<FilteredTodoBloc, FilteredTodoState>(
builder: (context, state) {
  print("building: Todo Item \$id " + key.toString());

  if (state is FilteredTodoLoadedState) {
  //retrieve the instance of the todo from the state using the id
    Todo t = (state).todos.firstWhere((element) => element.id == id);
    //use it in the InkWell widget as usual
    return InkWell(. . .);
        } else {
    return Row(
      children: const [
        Text("ERROR"),
      ],
    );
  }
});
```

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.

Source Code 2.119: Todo app - Bloc - using the *buildWhen* field to perform exclusive rebuild in te TodoView widget

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

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.

Source Code 2.120: Todo app - Bloc - using the *buildWhen* field to perform exclusive rebuild in te TodoItem widget

```
buildWhen: (previous, next) {
  if (next is FilteredTodoLoadedState &&
      previous is FilteredTodoLoadedState) {
      //check if the todo still exists
    if (next.todos.map((todo) => todo.id).toList().contains(id) == true) {
    //check if it is changed
      if (previous.todos.firstWhere((element) => element.id == id) ==
          next.todos.firstWhere((element) => element.id == id)) {
        return false;
      }
```

```
    } else {
      return false;
    }
  }
  return true;
}
```

Conclusions

Recap

	lines of code	time	classes
base functionalities	367	10-12 h	24
feature addition	67	15-20 m	2
rendering optimization	33	6-8 h	0

Table 2.3: Caption of the Table to appear in the List of Tables.

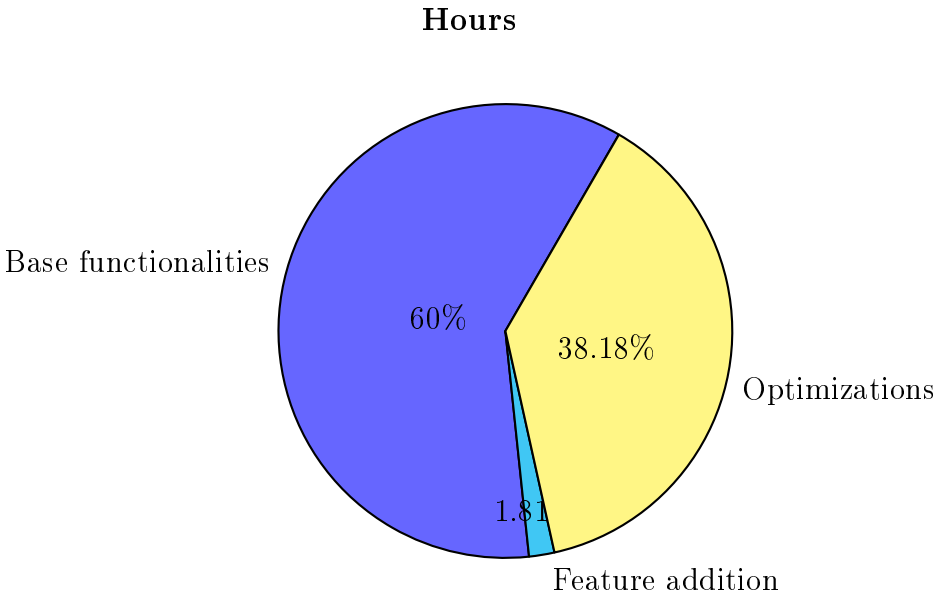


Figure 2.14: Caption of the Table to appear in the List of Tables.

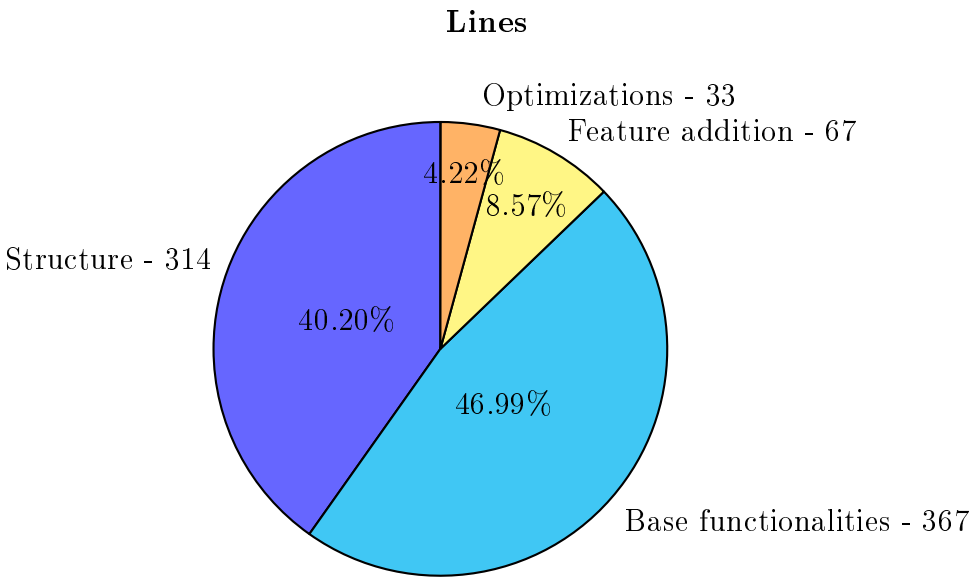


Figure 2.15: Caption of the Table to appear in the List of Tables.

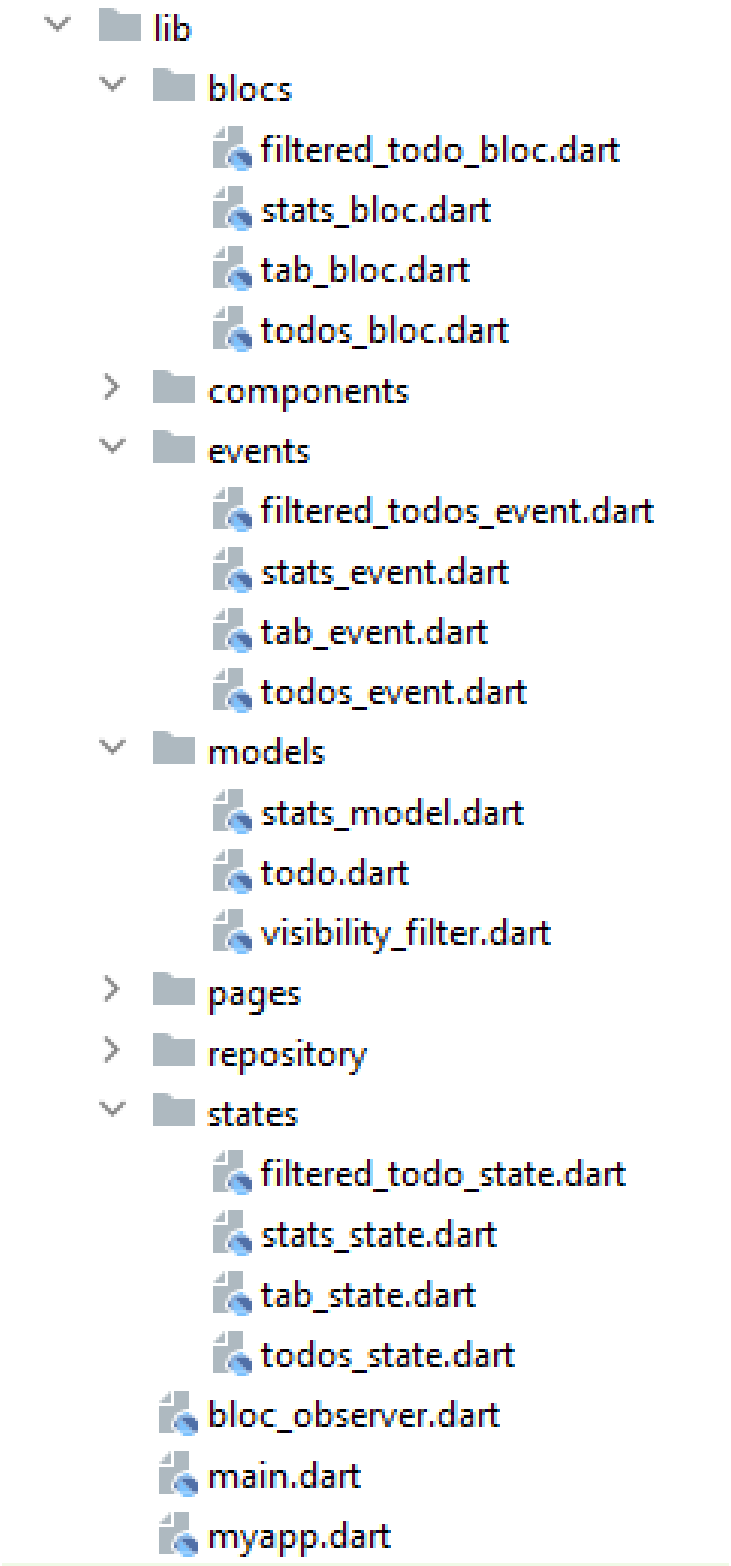
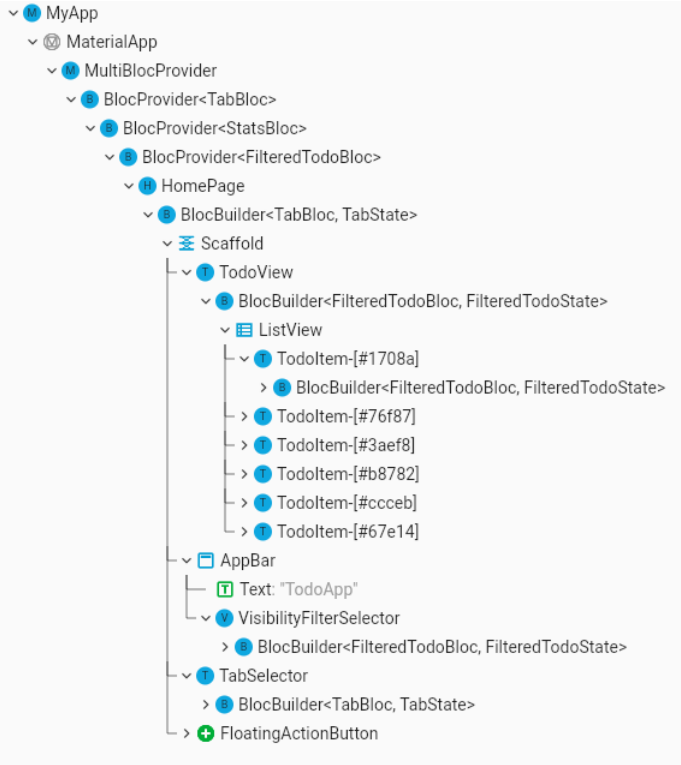
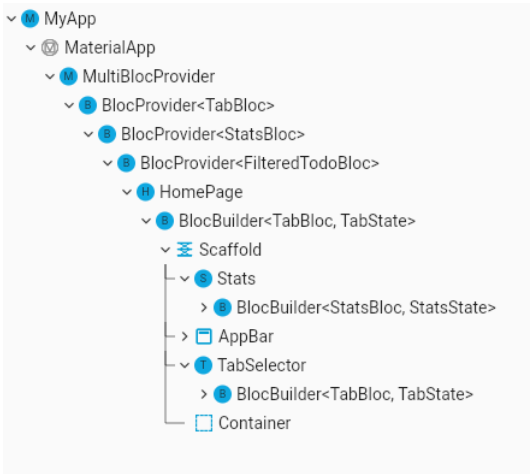


Figure 2.16: Show the tree structure after the FloatingActionButton in the HomePage is tapped.

Down below some images taken from an execution of the application. In this execution ,six todos are randomly created and only two of them are marked as completed.



(a) todos tab runtime UI.



(b) todos tab runtime widget tree.

Figure 2.17: Show the runtime Widget's tree and UI when visualizing todos tab.

Figure 2.21a shows how the application's UI looks like after few seconds from the start. Figure 2.21b show the widget's tree related with the run. Notice the TodoInheritedData widget as a child of the HomePage widget,it provides the state to the subtree.

2.2.5. MobX implementation

Here start the implementation process that utilizes the MobX state management solution.

Base funtionalities

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 changes in the observed variables. To be able to define observables is necessary to extend the intrested class 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 momentarily remembering that no rendering optimizations are 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.

The observable 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 syntax. The @observable annotation informs the code generator that those variables need to be made observable, moreover, the code generator automatically creates a getter and a setter action for them.

Source Code 2.121: Todo app - MobX- TodoStore abstract class definition

```
abstract class _TodoStore with Store {
  @observable
  List<Todo> todos = [];

  @observable
  VisibilityFilter filter = VisibilityFilter.all;
}
```

Actions - In this implementation the strict mode is set to "always". This choice reflects the common usage of the pattern. It is indeed a common choice to allow state mutations only through actions. This behavior comes with a lot of advantages that make it the correct choice for the major 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" 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.

Source Code 2.122: Todo app - MobX - TodoStore's changeFilter action definition

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

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.

Source Code 2.123: Todo app - MobX - TodoStore's setCompleted action definition

```
  @action
  void setCompleted(int id, bool completed) {
    assert(todoExists(id) != null, 'No todo with id : \$id');
    todos.where((element) => element.id==id).first.completed=completed;
  }

Todo? todoExists(int id) {
  List<Todo> result = todos.where((element) => element.id == id).toList();
  return result.isNotEmpty ? result.first : null;
}
```

The two usual methods to fetch and save the todos into the DataBase/repository are

implemented and annotated with the @action annotation too.

Source Code 2.124: Todo app - MobX - TodoStore's fetchTodos and saveTodos actions definitions

```
@action
Future<void> fetchTodos() async {
  todos = await TodoRepository.loadTodos();
}

@action
Future<void> saveTodos() async {
  await TodoRepository.saveTodos(todos);
}
```

Computed fields - computed fields are the part of the state that can be derived from other parts of the state. They are pivotal in the mobx state management solution because the package is able to smartly compute them and to perform lot of optimizations under the hood using 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*.

Source Code 2.125: Todo app - MobX - TodoStore's completedTodos and pendingTodos computed values definitions

```
@computed
List<Todo> get completedTodos =>
    todos.where((element) => element.completed).toList();
```

```
@computed
List<Todo> get pendingTodos =>
    todos.where((element) => !element.completed).toList();
```

pendingTodos and *completedTodos* methods are then used to implement other 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.

Source Code 2.126: Todo app - MobX - filteredTodos and stats computed value definition

```
@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();
}
```

The code generation - The _TodoStore class won't be directly used in the code but its actual implementation will. The _TodoStore is, indeed, an abstract class and is used by the code generator to implement the actual TodoStore class. A particular line of code must be placed just below the imports to allow the code generator to recognize the abstract class to implement. This line of code uses the "part" syntax followed by the name of the file to be generated. The generated code will be putted inside a file called counter.g.dart which is included with the part directive right below the imports. Without this line, the code generator will not produce any output. The generated file contains the _$TodoStore mixin. It is combined with the _TodoStore abstract class to finally implement the TodoStore.

Source Code 2.127: Todo app - MobX - TodoStore code generation

```
part 'todo_store.g.dart';

class TodoStore = _TodoStore with _\$TodoStore;
```

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

Source Code 2.128: Todo app - MobX - code generator directives

```
flutter pub run build_runner watch --delete-conflicting-outputs
```

It automatically generates 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 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.

The spy feature - 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 must be passed to the spy feature. The code inside this function is execute every time a action occurs. It is, indeed, 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.

Source Code 2.129: Todo app - MobX - spy feature attachment

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

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.

Making the state accessible - 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 todos can take place.

Source Code 2.130: Todo app - MobX - dispatch TodoStore using Provider widget

```
return Provider<TodoStore>(
    create: (_) => TodoStore()..fetchTodos(),
```

```
    child: MaterialApp(. . .),
    }));
```

The HomePage and the TabSelector component - The HomePage is almost entirely built using the state's 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 usage 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 VisibilityFilterSelector 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 should be rebuilt. In our case all components depends on the tab value and then every component is rebuilt once a tab change occurs. In cases some component are independent from the tab value and are wrapped into the observer widget they 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. The first solution is used.

Source Code 2.131: Todo app - MobX - HomePage implementation

```
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()
        );
      }
    );
  }
}
```

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 the tab value once the user taps on one specific BottomNavigationBarItem. Notice that the state change is wrapped into a *runInAction* object. This because if we 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 to "always". *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.

Source Code 2.132: Todo app - MobX - TabSelector component implementation

```
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: (...),
      );

  }
}
```

The VisibilityFilterSelector component - 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.

Source Code 2.133: Todo app - MobX - retrieving the TodoStore

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

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 *changeFilter* action previously defined is used to set the filter value to the tapped one.

Source Code 2.134: Todo app - MobX - VisibilityFilterSelector component implementation

```
return Observer(
  builder: (context) {
    print("building Visibilityfilter");

    return DropdownButton<VisibilityFilter>(
      value: store.filter,
```

```
      items: (...),
      onChanged: (tappedValue) {
        store.changeFilter(tappedValue!);
      },
    );
  },
);
```

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.

Source Code 2.135: Todo app - MobX - changing the state in the VisibilityFilterSelector without predefined action

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

I personally 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 a field is 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, indeed, brings way more meaning for the reader and prevents programmers to accidentally mutate the state in the implementation process. Even if I prefer the first approach, in the end , both approaches pass though actions in order 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 allow the mobx package to 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 with respect of the others. Consequently the entire state of the application is not well synchronized and the UI could reflect this inconsistency bringing to a bad situations. Mobx package ensure that actions,

and consequent reactions, are performed atomically without leaking any intermediate values as long as actions are used.

The Stats component - 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 .

Source Code 2.136: Todo app - MobX - Stats component implementation

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

The TodoView component and the TodoItem component - These two components represent the core of the UI development process. Remember now that no optimizations are considered for the moment. The TodoView widget needs to access the state and rerender once a change in the filteredList of todos occurs. In this scenario, the MobX pattern really shines. It is sufficient to wrap the entire TodoView 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 widget just acts as a visualizer for the corresponding todo and consequently takes as argument the todo instance to be visualized.

Source Code 2.137: Todo app - MobX - TodoView component implementation

```
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),
        );
      },
    );
  },
);
```

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

Source Code 2.138: Todo app - MobX - changing the state in the TodoItem component's Checkbox

```
onChanged: (value) {
  final store = Provider.of<TodoStore>(context);
  store.setCompleted(todo.id, value!);
  // the state could also be changed using
  // todo.completed = value!;
}),
```

The base functionalities are now working fine.

Features addition

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* arguments 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.

Source Code 2.139: Todo app - MobX - updateTodo and addTodo actions definition

```
@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();
}
```

Accessing these new actions in the UpdateTodoPage and in the AddTodoPage is simple by the fact that the Provider widget has been positioned as parent of the MaterialApp widget (from where different routes 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.

Source Code 2.140: Todo app - MobX - AddTodoPage implementation

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

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

Source Code 2.141: Todo app - MobX - UpdateTodoPage implementation

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

Rendering optimizations

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.

Source Code 2.142: Todo app - MobX - Todo model redefinition

```
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;
(...)
}
```

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.

Source Code 2.143: Todo app - MobX - Making the TodoItem responsive to state changes

```
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(...);
    });
  }
}
```

Conclusions

Recap

	lines of code	time	classes
base functionalities and understanding	92 (+106)	5-6 h	2 (+1)
feature addition	31 (+45)	10-15 m	0
rendering optimization	16 (+ 55)	30 m	1 (+1)

Table 2.4: Caption of the Table to appear in the List of Tables.

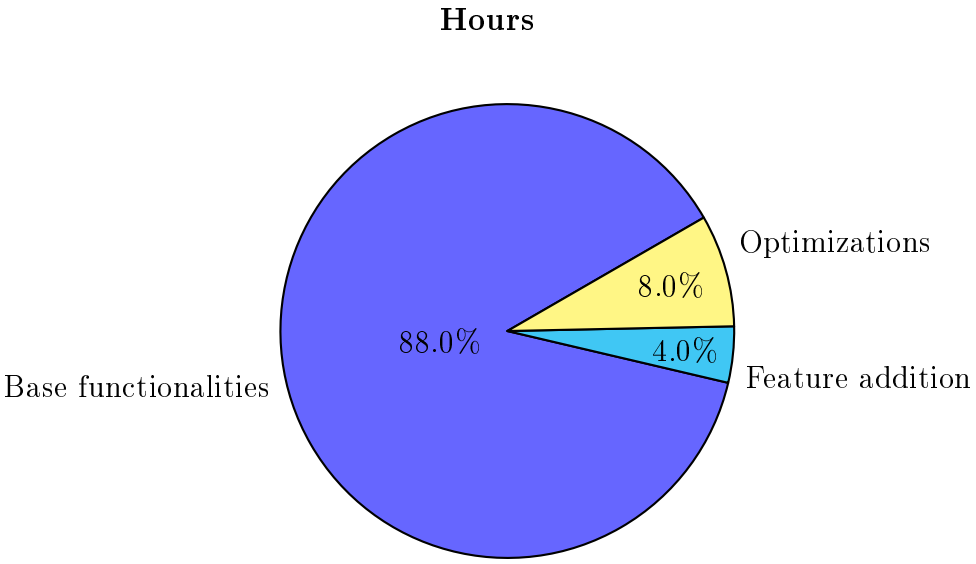


Figure 2.18: Caption of the Table to appear in the List of Tables.

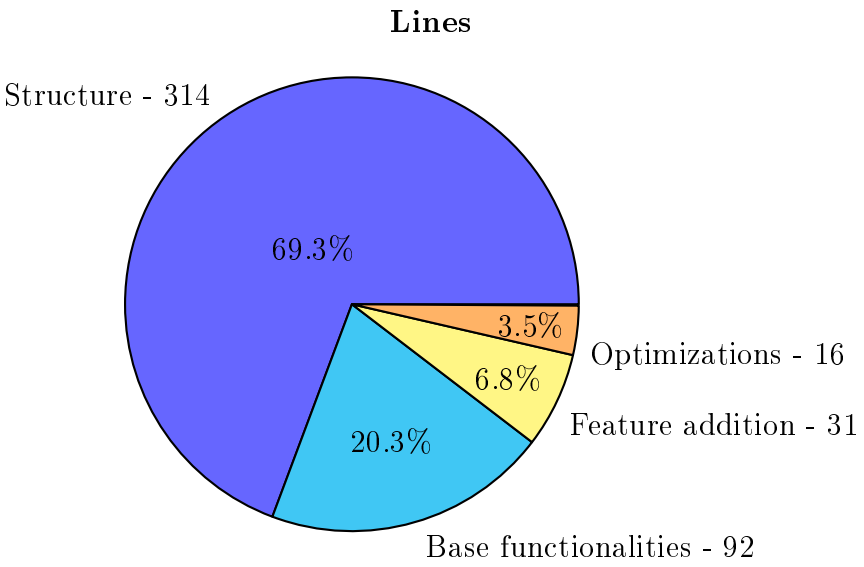


Figure 2.19: Caption of the Table to appear in the List of Tables.

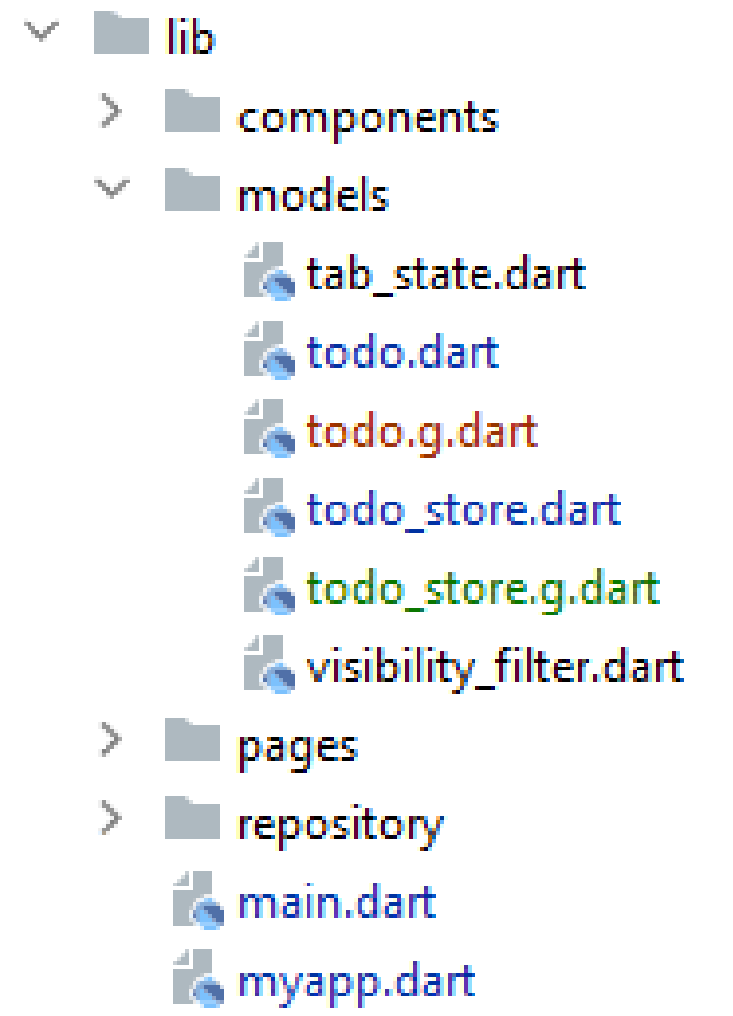
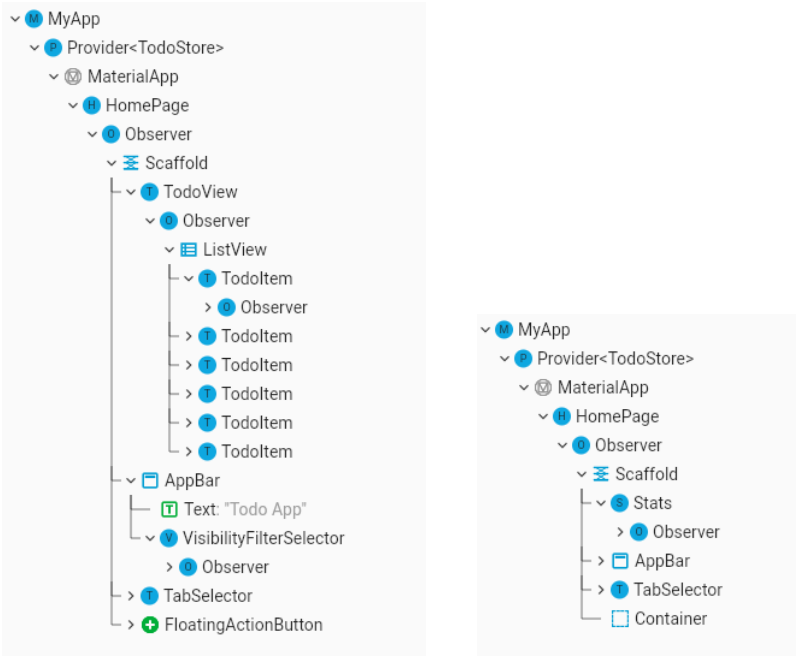


Figure 2.20: Show the tree structure after the FloatingActionButton in the HomePage is tapped.

Down below some images taken from an execution of the application. In this execution ,six todos are randomly created and only two of them are marked as completed.



(a) todos tab runtime UI.

(b) todos tab runtime widget tree.

Figure 2.21: Show the runtime Widget's tree and UI when visualizing todos tab.

Figure 2.21a shows how the application's UI looks like after few seconds from the start. Figure 2.21b show the widget's tree related with the run. Notice the TodoInheritedData widget as a child of the HomePage widget,it provides the state to the subtree.

3 | The Other app

Another app developed using same state managemnts solutions

4 | Comparisons

Some comparisons involving the data i kept and the other word file i have sent to you before

5 | Conslusions

Conclusions

A | Appendix A

If you need to include an appendix to support the research in your thesis, you can place it at the end of the manuscript. An appendix contains supplementary material (figures, tables, data, codes, mathematical proofs, surveys, ...) which supplement the main results contained in the previous chapters.

B | Appendix B

It may be necessary to include another appendix to better organize the presentation of supplementary material.

List of Figures

List of Tables

List of source codes

Acknowledgements

Here you might want to acknowledge someone.