



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

Abstract	i
Abstract in lingua italiana	iii
Contents	v
Introduction	1
1 State management solutions	3
1.1 SetState and InheritedWidget/InheritedModel	3
1.2 Redux	3
1.3 BLoc	3
1.4 MobX	3
1.5 GetX	3
2 The Todo app	5
2.1 General overview	5
2.1.1 Base functionalities	5
2.1.2 Adding new features	6
2.1.3 Renders optimization	6
2.2 Implementation	7
2.2.1 Shared project structure and files	7
2.2.2 Inherited widget/model and SetState implementation	21
2.2.3 Redux implementation	50
2.2.4 BloC implementation	50
2.2.5 MobX implementation	50
2.2.6 GetX implementation	50
3 The Other app	51

4	Comparisons	53
5	Conslusions	55
A	Appendix A	57
B	Appendix B	59
	List of Figures	61
	List of Tables	63
	List of source codes	65
	Acknowledgements	67

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. The application manages a list of todos. It is developed using the state management solutions proposed in Chapter 1. For every solution, three different development processes are taken. Moreover, a series of measurements, concerning the volume of the code and the effort, are collected.

2.1. General overview

This section explains in details the three development processes. These processes concern the implementation of the main functionalities, the addition of new ones and the performance optimization.

2.1.1. Base functionalities

This part of the development process aims to realize the skeleton of the app and the main functionalities. The output of the process will be an application that 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 *todo* tab and the *stats* tab. In the *todo* tab the list of todos is visualized. It is possible to filter todos using a `DropDownButton` in the top right corner inside the `AppBar`. The available filter values are:

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

The list of todos is visualized using a `TodoView` component. The elements that compose the list of todos are called `TodoItems`. `TodoItems` visualize the todo's name and description using two `Text` widgets and 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 it is possible to visualize the number of completed todos

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

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 process. This process is divided into two subparts. Both of them aims to add a single new feature.

The Add todo Feature

The first subpart adds the possibility to create new todos. It utilizes the FloatingActionButton , already present in the skeleton of the app in the bottom right corner, to push a new page called: AddTodoPage. In the AddTodoPage is possible to compile two TextField widget and use a TextButton widget to pop the page and create the new todo.

The Update feature

The second subpart adds the possibility to update existing todos. Tapping on a specific TodoItem the application navigates to a new page : the UpdateTodoPage. In the UpdateTodoPage is possible to compile two TextFields widgets and use a TextButton widget to pop the page and apply the modification.

2.1.3. Renders optimization

This part of the development process aims to perform some optimizations in terms of UI rendering and memory consumption. In particular, the code will be refactored in order to use the least UI renders possible and ,in other words, to call the least *build* methods 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. 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 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 ,instead, a non-structural change can rebuild only a subpart (the particular TodoItem). This because , when a structural change occurs, more than one TodoItem is affected and ,the most convinient way to mutate them all consistently ,is to rebuild the entire TodoView widget. Moreover, addind , deleting and substituting a TodoItem (and consequently add/delete/substitute a child to the TodoView tree node) is only possible 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 possible 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 even more fair , the code about the application's core and UI is shared between different solution's implementations. 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. And the same is for the UI. It uses the least widget and visual features possible. In the Figure 2.1 the shared folder's and file's structure is shown. Subsequent paragraphs explains how models, pages, components and the repository are implemented.

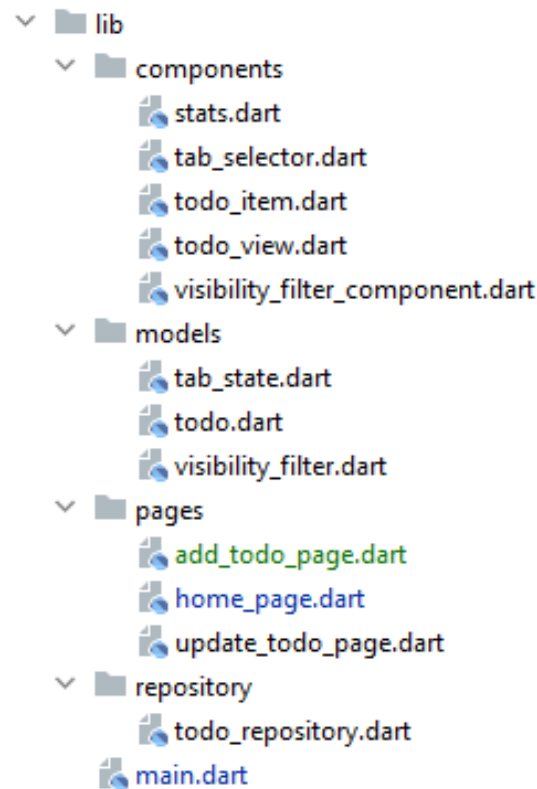


Figure 2.1: Todos app skeleton's folders structure.

The application's Root

The root widget of the application is called `MyApp`. It is a stateful widget composed by a `MaterialApp`. Inside the `MaterialApp`, three routes are defined : the `HomePage` , the `UpdateTodoPage` and the `AddTodoPage`. The *inicialRoute* is set to the `HomePage` as deafult. Inside the *main* function the `MyApp` widget is passed to the *runApp* method at the application's start.

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

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

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



```
@override
State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      initialRoute: "/",
      routes: {
        "/": (context) => const HomePage(),
        "/updateTodo": (context) => UpdateTodoPage(),
        "/addTodo": (context) => AddTodoPage(),
      },
    );
  }
}
```

Models and Repository

as show in code ?? HomePage's tabs are only two : *todos* and *stats*. In the *todos* tab todos are visualized. In the *stats* tab ,instead, some numerical recap of the todos is visualized. They are defined using an enumeration for simplicity.

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 of the Todo model matching every solution. Todo model ,indeed, change in different implementations. The sharable structure of the model ,however, can defined as below RIFERIMENTO. (

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;
}

```

The `TodoRepository` class simulate todos's fetching 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, populate a list with six new todos after the generation of their unique ID's. Subsequently, after 2 seconds, returns it to the caller.

Source Code 2.5: Todo app - `TodoRepository` definition

```

class TodoRepository {
    static Future<List<Todo>> loadTodos() async {
        Random rand = Random();
        List<Todo> todos = [];
        List<int> ids = [];
        while (ids.length < 6) {

```

```

        int newInt = rand.nextInt(1000)+2;
        if (!ids.contains(newInt)) {
            ids.add(newInt);
        }
    }
    todos = ids
        .map((number) => Todo(
            id: number,
            name: "Todo " + number.toString(),
            description: "description " + number.toString(),
            completed: rand.nextBool()))
        .toList();

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

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

```

Pages

Homepage uses a simple Scaffold widget. The AppBar contains a VisibilityFilterComponent only when the tab is set to *todos*. The body can change from *todos* tab to *stats* tab using the BottomNavigationBar (the TabSelector). 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 changing is implemented through setState but it will not be always the case. Also ,the HomePage, can be muted to Stateless widget in other implementations.).

Source Code 2.6: 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 Scaffold(
      appBar: AppBar(
        actions: [
          tab == TabState.todos
            ? const VisibilityFilterComponent()
            : Container()
        ],
        title: const Text("Todo App"),
      ),
      body: tab == TabState.todos ? const TodoView() : const Stats(),
      bottomNavigationBar: TabSelector(
        currTab: tab,
        onTabChange: ,
      ),
      floatingActionButton: tab == TabState.todos
        ? FloatingActionButton(
            child: const Icon(Icons.plus_one),
            onPressed: () {},
          ) : null,
    );
  }
}
```

The UpdateTodoPage uses a Scaffold widget. The body is filled with a Column with two TextFields and a TextButton inside. The TextButton is left empty for future implementation.

Source Code 2.7: Todo app - UpdatePage definition

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

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

  @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 desc'
        ),
    ),
    TextButton(onPressed: () {},
        child: const Text("Confirm")),
    ],
  ));
}

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

```

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

Source Code 2.8: Todo app - AddTodoPage definition

```

class AddTodoPage extends StatefulWidget {

  final void Function(String,String) addTodoCallback;

  const AddTodoPage({Key? key, required this.addTodoCallback}) : super(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: () {},
                        , child: const Text("Create"))
                ],
            ));
}

    @override
    void dispose() {
        textControllerName.dispose();
    }
}

```



```
        textControllerDesc.dispose();  
        super.dispose();  
    }  
}
```

Components

Components are widgets created with a specific aims. `TodoView` component take care to visualize a list of todos. Todos are accessed in different ways depending on the implementation. `TodoView` uses a `ListView` widget. *itemCount* and *itemBuilder* fields are left empty for future implementation.

Source Code 2.9: 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: ,  
            itemBuilder: (context, index) {  
                return TodoItem(  
  
                );  
            },  
        );  
    }  
}
```

TodoItem is a component that take care to visualize a specific todo. TodoItem is a stateless widget. It uses two Text widgets to display the todo's information and a Checkbox to change the todo's completion. It is wrapped in a InkWell widget to make is responsive to taps. Functions are left empty for future implementation.

Source Code 2.10: 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");
      },
      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) {}),
        ],
      ),
    );
  }
}
```

```

        ),
      );
    }
  }
}

```

TabSelector component provides a way to switch from tabs. Tabselector uses a BottomNavigationBar with as many BottomNavigationBarItems as TabState.values (in our case two). Functions's fields are left empty for future implementation.

Source Code 2.11: 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: ,
      onTap: (){},
      items: TabState.values
        .map((tab) => BottomNavigationBarItem(
          label: describeEnum(tab),
          icon: Icon(
            tab == TabState.todos ? Icons.list : Icons.show_chart,
          ),
        ))
        .toList(),
    );
  }
}

```

```

    );
  }
}

```

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

Source Code 2.12: 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:,
      items: VisibilityFilter.values.map((filter) {
        return DropdownMenuItem<VisibilityFilter>(
          child: Text(describeEnum(filter)), value: filter);
      }).toList(),
      onChanged: (filter) {

      },
    );
  }
}

```

Stats component takes care to visualize some numerical representation of the list of todos.

Stats component is a Stateless widget composed by Text widget showing stats value.

Source Code 2.13: 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());  
  }  
}
```

2.2.2. Inherited widget/model and SetState implementation

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

State management solution's introduction

setState method notify the framework that the internal state of this object has changed. Whenever you change the internal state of a State object, make the change in a function that you pass to *setState*.

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

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

being set.

Calling *setState* notifies the framework that the internal state of this object has changed in a way that might impact the user interface in this subtree, which causes the framework to schedule a build for this State object.

If you just change the state directly without calling *setState*, the framework might not schedule a build and the user interface for this subtree might not be updated to reflect the new state.

Inherited widget are a base class for widgets that efficiently propagate information down the tree. To obtain the nearest instance of a particular type of inherited widget from a build context, use *BuildContext.dependOnInheritedWidgetOfExactType*. Inherited widgets, when referenced in this way, will cause the consumer to rebuild when the inherited widget itself changes state. The convention is to provide a static method *of* on the *InheritedWidget* which does the call to *BuildContext.dependOnInheritedWidgetOfExactType*. This allows the class to define its own fallback logic in case there isn't a widget in scope. An *InheritedWidget* is not intended to be used as the base class for models whose dependents may only depend on one part or "*aspect*" of the overall state. Indeed inherited widget's dependents are unconditionally rebuilt when the inherited widget changes.

InheritedModel widget is similar except that dependents aren't rebuilt unconditionally. Widgets that depend on an *InheritedModel* qualify their dependence with a value that indicates what "*aspect*" of the model they depend on. When the model is rebuilt, dependents will also be rebuilt, but only if there was a change in the model that corresponds to the aspect they provided.

Base 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 2.14: 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 will contain the todos matching the filter. Inside the constructor, final variables are initialized with their corresponding arguments and, *stats* and *filteredTodos* variables, are computed.

Source Code 2.15: 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,
      filteredTodos = filterTodo(todos, filter),
      super(child: child, key: key);
}
```

filterTodos function is just a function that takes the full list of todos and a filter and returns the filtered list. Important to notice 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* function must be overridden inside the `TodoInheritedData` class. This function 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 to call the *updateShouldNotify* method and to decide 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 2.16: 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 are the same. In the code above, 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 that can be called without instantiating 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 (of the provided type) is found it raises a runtime error.

Source Code 2.17: 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 application in a given moment. It cannot change its internal values neither substitute itself with another instance. In practice , what happens, is that a stateful widget is created. This stateful widget will contain the state and will bother to create a new instance of the `TodoInheritedData` widget containing it. Everytime its internal state is changed (using `setState`) a new instance of `TodoInheritedData` widget is produced and substituted with the old one. In this way ,changes are reported to the subtree. The subtree sees a different image of the state and reacts to it. Personally, I did not appreciate this approach `InheritedWidget` uses. On one way it is simple and works really well for its purpose , on the other 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 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 synchronized. A bad sincronization 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 synchronize 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 will be used with a purpose that goes behind its real usage. Anyway, it is possible that different widgets sees different screenshots of the data and the bigger the application grow the higher will be the probability that this happens. 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 `TodoInheritedData` creation)

Source Code 2.18: 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` *filter*, 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 2.19: Todo app - InheritedWidget - TodoProvider 's init method implementation

```

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

```

loadTodos is a *TodoRepository*'s async function that simulate the retrieval of the todos from a database as defined in the paragraph 2.2.1.

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 instance of *TodoInheritedData* is accessible only in a context where a *TodoProvider* is already present. In other word *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 with the *TodoProvider* widget.

Source Code 2.20: Todo app - InheritedWidget - data's injection in the tree

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

The *TodoView* component

TodoView component can now be populated. Its implementation can be found in paragraph 2.2.1 in subsection 2.2.1. It is a stateless widget that looks up for the *filteredTodos* list, contained in the *TodoInheritedData* 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 2.21: Todo app - InheritedWidget - *TodoView* implementation

```
class TodoView extends StatelessWidget {

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

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

```

print("Building TodoView");

final List<Todo> filteredTodos = TodoInheritedData.of(context).filteredTodos;

return ListView.builder(
  itemCount: filteredTodos.length,
  itemBuilder: (context, index) {
    return TodoItem(
      todo: filteredTodos.elementAt(index),
    );
  },
);
}
}

```

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, an empty page appears (todos are empty at the beginning) and then, after a 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 `VisibilityFilterComponent` is ready to be used as defined in `RIFERIMENTO`. Its `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 2.22: 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");
  VisibilityFilter filter= TodoInheritedData.of(context).filter;
  return DropdownButton<VisibilityFilter>(
    value: filter,
    items: VisibilityFilter.values.map((filter) {
      return DropdownMenuItem<VisibilityFilter>(
        child: Text(describeEnum(filter)), value: filter);
    }).toList(),
    onChanged: (filter) {

    },
  );
}
}

```

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, in its 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 other 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 completely 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 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 practice, a new function, called *onChangeFilter*, is added inside the *TodoProvider* widget. This function takes a *VisibilityFilter* value as parameter and set the value of *TodoProvider*'s *filter* variable using *setState* method.

Source Code 2.23: 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 consequence 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, the *onChangeFilter* function must be provided to the *TodoInheritedData* widget to make it accessible in the subtree. A new parameter is added in the *TodoInheritedData* class, though.

Source Code 2.24: Todo app - *InheritedWidget* - *TodoInheritedData* widget expansion

```
class TodoInheritedData extends InheritedWidget {  
  {...}  
  final void Function(VisibilityFilter) onChangeFilter;  
  {...}
```

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

Source Code 2.25: Todo app - InheritedWidget -onChangeFilter function injection into TodoInheritedData widget

```
@override
Widget build(BuildContext context) {
  return TodoInheritedData(
    todos: todos,
    onChangeFilter: onChangeFilter,
    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 2.26: 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 at the moment. It takes as parameter a Todo instance and takes care of displaying it as defined in source code 2.4. TodoItem does not read 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, inside every TodoItem, is just showing their todo's completion. Its *onChange* function is still empty and, for this reason, it does nothing

when tapped. When the `CheckBox` is tapped a change in the corresponding `Todo`'s *completed* field should be fired and a rebuild of the `TodoItems` performed. In order to do so, `TodoInheritedData` widget should provide a state changing function that allow to perform this change. The process to be followed and the reasoning is the same exposed in the previous paragraph 2.2.2 when we spoke about the `VisibilityFilterSelector` widget. Going back to the `TodoProvider` stateful widget a function ,called *onSetCompleted* , is added . This function takes as parameter the *id* of the `todo` to be changed and the new value for the *completed* field.

Source Code 2.27: Todo app - `InheritedWidget` - `TodoProvider` widget *onSetCompleted* function implementation

```
void onSetCompleted(int id, bool completed) {
  assert(todoExists(id) != null, 'No todo with id : \'$id\');

  setState(() {
    todos = todos.map((e) {
      if (e.id == id) {
        return Todo(
          id: id,
          name: e.name,
          description: e.description,
          completed: completed);
      } else {
        return e;
      }
    }).toList();
  });
}
```

In the *onSetCompleted* function , *todos* list is scanned using the *map* method. Once the `todo`, with the corresponding *id* ,is found, its *completed* value is updated to the new value. Calling the *onChangeFilter* method on the `TodoProvider` stateful widget will cause the *build* method to run again and to create another `TodoInheritedData` widget. As usual , the function is passed from the `TodoProvider` widget to the `TodoInheritedData` widget ,on its creation. In this way, the function is accessible down the tree. It is now possible

to call it inside the `onChanged` method of the `TodoItem`'s `Checkbox`.

Source Code 2.28: Todo app - `InheritedWidget` - `TodoItem`'s `Checkbox` *onChanged* field implementation

```
Checkbox(  
  value: todo.completed,  
  onChanged: (value) {  
    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 state, the part concerning the stats. The nearest instance of the `TodoInheritedData` widget is retrieved using the *of* method and used to fill the `Text` widget.

Source Code 2.29: 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: Text(TodoInheritedData.of(context).stats.toString());  
    );  
  }  
}
```

```
    }
}
```

The TabSelector component

The part of the state concerning the tab includes just 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 the first choice 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 and do the job well. Sometimes, indeed, for the parts of the state that can be referred as "local" ,meaning that 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. For example, 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 practice, to create a local variable and handle it with *setState* and stateful widgets instead of adding a new variable to the TodoInheritedData widget , handle 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 function ,in order to change the tab value, to implement the feature. This new function ,called *onTabChange*, takes a int value as parameter and uses the *setState* method to update the *tab* variable. (it takes as argument a int ,and not a TabState ,because the *onTap* field of the BottomNavigationBar, in the TabSelector component ,provides an int. This int value refers to the index of the tapped element inside the list of BottomNavigationBarItem widgets we provided to the *items* field.

Source Code 2.30: 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 ,to the *TabSelector* widget ,as parameter and use it in the *BottomNavigationBar*'s *onTap* field.

Source Code 2.31: Todo app - *InheritedWidget* - *TabSelector* component implementation

```
class TabSelector extends StatelessWidget {
  final TabState currTab;
  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");

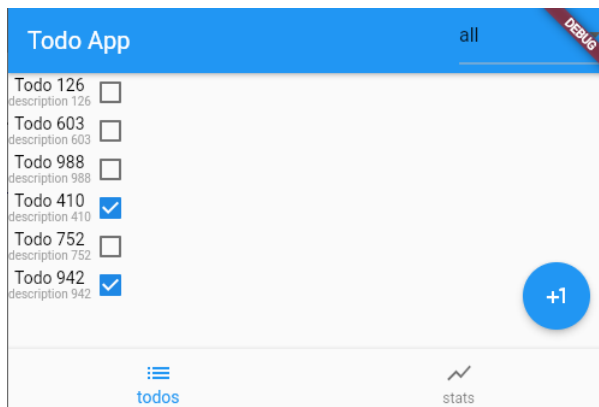
    return BottomNavigationBar(
      currentIndex: TabState.values.indexOf(currTab),
      onTap: onTabChange, //used here
      items: TabState.values
        .map((tab) => BottomNavigationBarItem(
          label: describeEnum(tab),
          icon: Icon(
            tab == TabState.todos ? Icons.list : Icons.show_chart,
          ),
        ))
        .toList(),
    );
  }
}
```

```
}
}
```

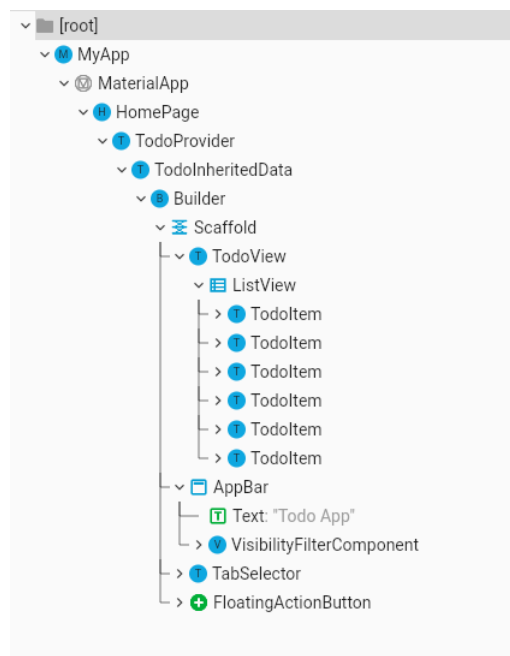
It is now possible to switch from tabs.

Summary

The whole process was fast and straight forward. 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 2.2: Show the runtime Widget's tree and UI when visualizing todos tab.

Figure 2.2a shows how the application's UI looks like after few seconds from the start. Figure 2.2b 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.



Figure 2.3: Show the runtime Widget's tree and UI when visualizing stats tab.

Figure 2.3a shows how the application UI looks like after the user taps on the TabSelector's stats button. Figure 2.3b show the widgets tree related with the run after the button is clicked.

Time spent: 2-3 hours

Lines of code written/updated: 86

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

Features addition

Here stats the development process where the todo-addition feature and the todo-update feature are implemented.

Todo addition feature

First thing first, TodoInheritedData widget must provide a function to add todos. A new function is created in the TodoProvider widget and passed to the TodoInheritedData widget. This new function will be called *onAddTodo* and will take two parameters (name and description).

Source Code 2.32: Todo app - InheritedWidget - TodoProvider *onAddTodo* function implementation

```
void onAddTodo(String name, String desc) {
    Random rand = Random();
```

```

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

```

After generating a new unique id ,it creates a new Todo object called *newTodo* with the *completed* field set to *false* . Adding the new Todo ,to the TodoProvider's state *todos* list , requires a bit of workaround. The state of a stateful widget is immutable. Yes, it is a bit counterintuitive but ,as we already said , in functional programming, is generally better to create new instances instead of mutating existing one. Indeed, stateful widget's state can only be changed by the *setState* method. Unfortunately, the method *add* for lists, dart language provides, is of type void and do not return a new list but add the new value to the existing list's instance. For this reason ,directly calling the *add* method to the TodoProvider's local lists *todos* will have no effect. That list is immutable and cannot be changed. TodoProvider's *todos* list must be completely replaced with a new list containing also the new todo. First ,a new temporary list ,called *newList*, is created and populated with the 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 2.33: Todo app - InheritedWidget - onAddTodo function propagation
mbox

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

```
@override
```

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

```
mbox
```

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 2.4. The `AddTodoPage` is added as a child of the `MaterialApp` widget.

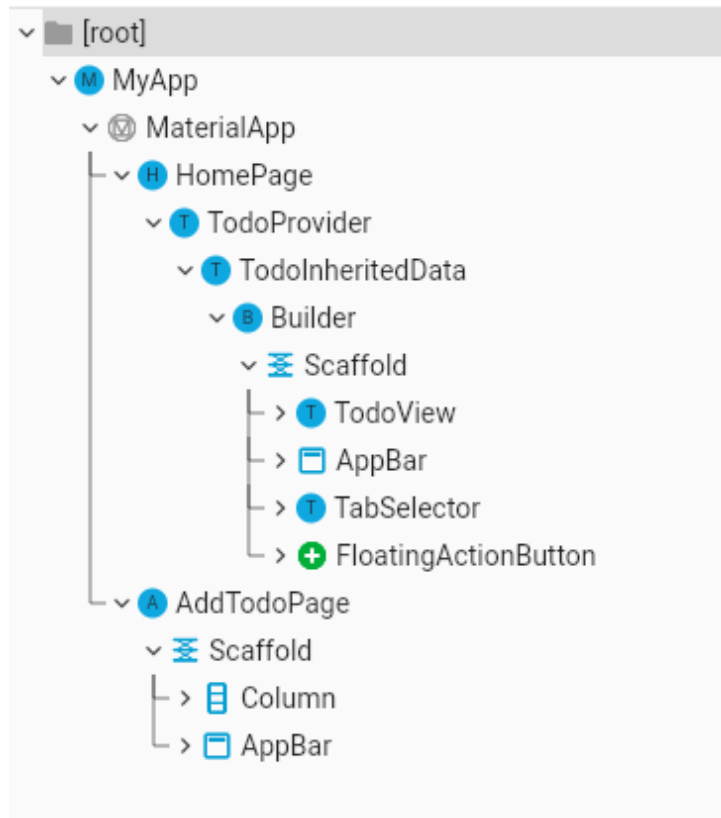


Figure 2.4: Show the tree structure after the FloatingActionButton in the HomePage is tapped.

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, though, 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');
```

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

Source Code 2.34: Todo app - InheritedWidget - AddTodoPage's callback function parameter creation


```
class AddTodoPage extends StatefulWidget {

    final void Function(String,String) addTodoCallback;
    { . . . }
```

mbox

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 2.35: Todo app - InheritedWidget - MaterialApp AddTodoPage route ridefinition

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

The only part to fill in, in the AddTodoPage, is the *onPressed* field of the TextButton. The callback function is then used, inside it, to add the new todo to the list, then, the AddTodoPage is popped.

Source Code 2.36: Todo app - InheritedWidget - AddTodoPage TextButton onPressed field implementation

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

When the AddTodoPage is popped, the new Todo is added and the HomePage is rebuilt.

Time spent: 20-30 minutes

Lines of code written/updated: 24

Classes/widget created: 0

Todo updating feature

A new function must be implemented ,in the `TodoProvider` widget ,and passed to the `TodoInheritedData` widget. This new function will be called *onUpdateTodo* and takes three arguments: the *id* of the todo to be updated, the *newName* that should be set and the *newDesc*.

Source Code 2.37: Todo app - InheritedWidget - todoProvider's onUpdateTodo function implementation

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

onUpdateTodo function first checks if a todo matching the id exists. Then, for the same immutability concept we dealt with in paragraph 2.29, a *newTodosList* is created and populated with the elements inside 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 and passing the method to id.

Source Code 2.38: Todo app - InheritedWidget - onUpdateTodo function propagation

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

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

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 will be of type *Function* taking two *Strings* as arguments (the id will be already set up by the calling page).

Source Code 2.39: Todo app - InheritedWidget - UpdateTodoPage callback variable creation

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

All is 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 only 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 2.40: 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 2.41: Todo app - InheritedWidget - onUpdateTodo function propagation

```
Navigator.pushNamed(context, "/updateTodo",
  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 parameter (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 2.42: Todo app - InheritedWidget - onUpdateTodo function propagation

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

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

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

```
TextButton(onPressed: () {

  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.

Time spent: 20-30 minutes

Lines of code written/updated: 43

Classes/widget created: 1 for arguments between routes

Render optimizations

The optimizations performed in this part are explained more in details in paragraph RIFERIMENTO RENDERS OPTIMIZATION. 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 interesting, though, 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`. `InheritedWidget`, 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 access 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 achieve 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.

Proceeding with the optimization process, first thing to do is to substitute the extension to `InheritedWidget` with the `InheritedModel` one, in the `TodoInheritedData` class.

Source Code 2.44: 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. Moreover, 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 2.45: Todo app - `InheritedModel` - *of* method implementation

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

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 *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 *updateShouldNotifyDependent* that is just like the `InheritedWidget`’s one, *updateShouldNotify* , but takes as argument also a Set of ints called *dependencies* (aspects). This method is called once for every widget that registered to state changes. The *dependencies* variable will contain all aspects the widgets registered to (only one for widget in our case).

Source Code 2.46: Todo app - `InheritedModel` - *updateShouldNotifyDepented* method implementation

```
@override
```

```

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

```

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

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

```

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

```



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

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 optimization, `TodoItem` widgets rebuilt correctly comes from the fact that the transition of the state caused the entire `TodoView` widget to rebuild. The consequence was that `TodoItems` were destroyed 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 synchronization 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 2.48: Todo app - `InheritedModel` - `TodoItem` widget todo look up

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

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

Time spent: 8-10 hours

Lines of code written/updated: 49

Classes/widget created: 0

2.2.3. Redux implementation

2.2.4. BloC implementation

2.2.5. MobX implementation

2.2.6. GetX implementation

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

2.1	Todos app skeleton's folders structure.	8
2.2	Show the runtime Widget's tree and UI when visualizing todos tab.	36
2.3	Show the runtime Widget's tree and UI when visualizing stats tab.	37
2.4	Show the tree structure after the FloatingActionButton in the HomePage is tapped.	40

List of Tables

List of source codes

2.1	Todo app - MaterialApp and main function definition	8
2.2	Todo app - TabState model definition	9
2.3	Todo app - VisibilityFilter model definition	10
2.4	Todo app - Todo model definition	10
2.5	Todo app - TodoRepository definition	11
2.6	Todo app - HomePage definition	12
2.7	Todo app - UpdatePage definition	14
2.8	Todo app - AddTodoPage definition	15
2.9	Todo app - TodoView definition	17
2.10	Todo app - TodoItem definition	18
2.11	Todo app - TabSelector definition	19
2.12	Todo app - VisibilityFilterSelector definition	20
2.13	Todo app - Stats definition	21
2.14	Todo app - InheritedWidget - extension to InheritedWidget	22
2.15	Todo app - InheritedWidget- TodoInheritedData implementation	23
2.16	Todo app - InheritedWidget -updateShouldNotify method override	24
2.17	Todo app - InheritedWidget - TodoInheritedData of method override	24
2.18	Todo app - InheritedWidget - TodoProvider implementation	25
2.19	Todo app - InheritedWidget - TodoProvider 's init method implementation	26
2.20	Todo app - InheritedWidget - data's injection in the tree	27
2.21	Todo app - InheritedWidget - TodoView implementation	27
2.22	Todo app - InheritedWidget - VisibilityFilterComponent implementation	28
2.23	Todo app - InheritedWidget - TodoProvider's onChangeFilter implementation	30
2.24	Todo app - InheritedWidget - TodoInheritedData widget expansion	30
2.25	Todo app - InheritedWidget -onChangeFilter function injection into TodoInheritedData widget	30
2.26	Todo app - InheritedWidget - DropdownButton's onChanged field implementation	31

2.27	Todo app - InheritedWidget - TodoProvider widget <i>onSetCompleted</i> function implementation	32
2.28	Todo app - InheritedWidget - TodoItem's Checkbox <i>onChanged</i> field implementation	33
2.29	Todo app - InheritedWidget - Stats component implementation	33
2.30	Todo app - InheritedWidget - HomePage's <i>onTabChange</i> function implementation	34
2.31	Todo app - InheritedWidget - TabSelector component implementation . . .	35
2.32	Todo app - InheritedWidget - TodoProvider <i>onAddTodo</i> function implementation	37
2.33	Todo app - InheritedWidget - <i>onAddTodo</i> function propagation	38
2.34	Todo app - InheritedWidget - AddTodoPage's callback function parameter creation	40
2.35	Todo app - InheritedWidget - MaterialApp AddTodoPage route ridefinition	41
2.36	Todo app - InheritedWidget - AddTodoPage TextButton <i>onPressed</i> field implementation	41
2.37	Todo app - InheritedWidget - todoProvider's <i>onUpdateTodo</i> function implementation	42
2.38	Todo app - InheritedWidget - <i>onUpdateTodo</i> function propagation	43
2.39	Todo app - InheritedWidget - UpdateTodoPage callback variable creation .	43
2.40	Todo app - InheritedWidget - <i>onUpdateTodo</i> function propagation	44
2.41	Todo app - InheritedWidget - <i>onUpdateTodo</i> function propagation	44
2.42	Todo app - InheritedWidget - <i>onUpdateTodo</i> function propagation	45
2.43	Todo app - InheritedWidget - <i>onUpdateTodo</i> function propagation	45
2.44	Todo app - InheritedModel - extension to InheritedModel	46
2.45	Todo app - InheritedModel - of method implementation	47
2.46	Todo app - InheritedModel - <i>updateShouldNotifyDepended</i> method implementation	47
2.47	Todo app - InheritedModel - <i>updateShouldNotifyDepended</i> method pseudocode	48
2.48	Todo app - InheritedModel - TodoItem widget todo look up	50

Acknowledgements

Here you might want to acknowledge someone.

