POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

State Management in Flutter

Tesi di Laurea Magistrale in
Computer Science - Ingegneria Informatica

Author: **Lorenzo Ventura**

Student ID: 906003
Advisor: Prof. Luciano Barese
Co-advisors: Name Surname, Name Surname
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 dfndn

1 | State management solutions

sdfds sddgdsgddfew dsfsdfsd

1.1. SetState and InheritedWidget/InheritedModel

sdfsdf

1.2. Redux

dfdsds

1.3. BLoc

sdfsdfs

1.4. MobX

sdfsdfs

1.5. GetX

sdfsdf

2 | The Todo app

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

2.1. General overview

The development process is divided in three parts.

2.1.1. Base functionalities

It offers the possibility to visualize and partially handle todos. It is composed of a single page: the HomePage. The HomePage is composed by an appbar and two tabs: the todo tab and the stats tab. In the todo tab the list of todos is visualized. Is possible to filter the todo using a DropdownButton in the top right corner inside the AppBar. The possible filter values are:

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

The elements inside the list of todos are called TodoItems. TodoItems visualize the todo's name and description using a Text widget and completion using a Checkbox. It is possible to use the checkbox to mark a Todo as completed or to mark it as pending. In the stats Tab instead is possible to visualize the number of completed todos through a Text widget. In the lower part a TabSelector allow to switch from tabs.

2.1.2. Adding new features

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

The Add todo Feature

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

The Update feature

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

2.1.3. Renders optimization

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

2.2. Implementation

sdfdsggsd

2.2.1. Shared project structure and files

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

Main

The main function will run the root widget of the app called MyApp. It is a stateful widget composed by a MaterialApp. Inside the MaterialApp three roots are defined. The HomePage , the UpdateTodoPage and the AddTodoPage. The inizialRoute is set to the HomePage.

```
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

First model presented is the TabState model. It represent the Tab visualized in the HomePage. TabState are only two : todos and stats. In the todos tab todos are visualized instead in the stats tab some numerical recap of the todos is visualized.

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

Possible filters for the list of todos are enumerated in the VisibilityFilter enumeration.

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

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

```
@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;
}
```

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

```
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 to stats tab using the bottomNaviagationBar TabSelector. An empty FloatingActionButton is present for future implementation. (note: some small pieces could change in different solution's implementation. in the above example the tab changing is implemented through setState but it will not be always the case. Also the HomePage can be muted to Stateless widget in other implementations.).

```
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 composed by a Column with inside a TextField and a TextButton. The TextButton is left empty for future implementation.

```
class UpdateTodoPage extends StatefulWidget {
  final Todo todo;

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

  @override
  State<UpdateTodoPage> createState() => _UpdateTodoPageState();
}

class _UpdateTodoPageState extends State<UpdateTodoPage> {
  final textController = TextEditingController();

  @override
  Widget build(BuildContext context) {

    return Scaffold(
        appBar: AppBar(
          title: const Text("Update Todo"),
        ),
        body: Column(
          children: [
            TextField(
              controller: textController,
              decoration: const InputDecoration(
                  border: OutlineInputBorder(), hintText: 'Enter a new name'),
            ),
            TextButton(onPressed: (){}, child: const Text("Confirm"))
          ],
        ));
  }

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

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

```
class AddTodoPage extends StatefulWidget {

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

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

class _UpdateTodoPageState extends State<AddTodoPage> {
  final textControllerName = TextEditingController();
  final textControllerDesc = TextEditingController();

  @override
  Widget build(BuildContext context) {

    return Scaffold(
        appBar: AppBar(
          title: const Text("Add Todo"),
        ),
        body: Column(
          children: [
            TextField(
              controller: textControllerName,
              decoration: const InputDecoration(
                  border: OutlineInputBorder(), hintText: 'Enter a name'),
            ),
            TextField(
              controller: textControllerDesc,
              decoration: const InputDecoration(
                  border: OutlineInputBorder(), hintText: 'Enter a description'),
            ),
            TextButton(onPressed: (){},
 child: const Text("Create"))
          ],
        ));
  }

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

Components

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

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

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

```
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 DropwodnMenuItems as VisibilityFilter.values (in our case two). Function fields are left empty for future implementation.

```
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 is only a Text widget showing stats value.

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

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

    return 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 *onChangeFilter* .

```
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 *onChangeFilter* 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. In the example above, the value returned will be null in that case, but it could also have defaulted to a value. An InheritedWidgets that's intended to be used as the base class for models whose dependents may only depend on one part or "*aspect*" of the overall model. An inherited widget's dependents are unconditionally rebuilt when the inherited widget changes.
InheritedModel widget is similar except that dependents aren't rebuilt unconditionally. Widgets that depend on an InheritedModel qualify their dependence with a value that

indicates what "*aspect*" of the model they depend on. When the model is rebuilt, dependents will also be rebuilt, but only if there was a change in the model that corresponds to the aspect they provided.

Base App

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

```
class TodoInheritedData extends InheritedWidget{
```

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

```
class TodoInheritedData extends InheritedWidget{
  final List<Todo> todos;
 final List<Todo> filteredTodos;
 final VisibilityFilter filter;
 final int stats;

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

Is important to understand that TodoInheritedData widget is stateless. It cannot be changed (every value is final) but instead a new TodoInheritedData widget must be provided when a data change occurs. The *updateShouldNotify* function must be overridden inside TodoInheritedData to avoid ui rebuilding when a new state without actual data changes occurs. Once a TodoInheritedData element is replaced with a new one this new element will take care to call the *updateShouldNotify* function to decide whether is necessary or not to notify changes in the subtree. If the function returns *true* the subtree is rebuilt, if return *false* instead is not.

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

In our case the *listEquals* function takes as parameters the old *filteredTodos* list and the new one and compare them element by element checking if changes were made. In the particular case no changes were performed it returns *true* and will lead the *updateShouldNotify* function to return *false* and not to rebuild the entire subtree. At this point our TodoInheritedData can be used it in a stateful widget.

```
class TodoProvider extends StatefulWidget {
  const TodoProvider({Key? key, required this.child}) : super(key: key);

  final Widget child;

  @override
  _TodoProviderState createState() => _TodoProviderState();
}

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

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

Note that the VisibilityFilter *filter*is set as all by default as convention. We add also an *init* method to fetch the data from the repository on widget's creation.

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

loadTodos is a TodoRepository's async function that simulate the retrieval of the todos from a database. We need to declare also the *of* method to retrieve our TodoInheritedData down the tree. This method just extracts the nearest TodoInheritedData element up in the tree using *dependOnInheritedWidgetOfExactType* method.

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

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

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

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

```
class TodoView extends StatelessWidget {

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

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

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

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

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

```
class TodoItem extends StatelessWidget {
  final Todo todo;

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

  @override
  Widget build(BuildContext context) {
    return Row(
        children: [
          Column(
            children: [
              Text(todo.name,
                  style: const TextStyle(fontSize: 14, color: Colors.black)),
              Text(todo.description,
                  style: const TextStyle(fontSize: 10, color: Colors.grey)),
            ],
          ),
          Checkbox(
              value: todo.completed,
              onChanged: (value) {
              }),
        ],
      ),
    );
  }
}
```

At this point we got a single page (Homepage) that contains a TodoView showing *filteredTodos* list's todos contained in the TodoInheritedData inside a TodoProvider widget. When the application starts we first see and empty page (todo are empty at the beginning) and then after few seconds a list of todos with their names, descriptions and completions appears. A list of fitered todos can be visualized but is not interactable yet. In the app HomePage's AppBar we already set up a VisibilityFilterComponent that is nothing else than a stateless widget. In its build method a DropdownButton's value field is set up looking up for the *filter* values in the TodoInheritedData. Then the *items* field is filled with a list of DropdownMenuItem that comes from the mapping of all possible VisibilityFilter values to DropdownMenuItems.

```
class VisibilityFilterComponent extends StatelessWidget {

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

      },
    );
  }
}
```

For what concerns the *onChanged* field a function that takes as single parameter a filter value must be provided. In particular we want this function to change the state contained in the TodoInheritedData (the *filter* variable) and to fire a rebuild of the TodoInherited-Data subtree. As we mentioned above TodoInheritedData contains only final fields and should never be modified. Instead, a new TodoInheritedData element should be created in the TodoProvider build method with the modified data. In the to TodoProvider.dart a function called *onChangeFilter* is added. This function takes the new *filter* values as parameter and changes the value of the *filter* in the stateful widget calling *onChange-Filter* . Doing so the build function is called again with the new filter value and a new TodoInheritedData widget is created.

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

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

```
@override
```

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

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

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

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

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

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

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

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

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

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

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

    return Text(TodoInheritedData.of(context).stats.toString());
  }
}
```

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

```
TabState tab = TabState.todos;
```

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

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

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

  const TabSelector(
      {Key? key, required this.currTab, required this.onTabChange})
      : super(key: key);

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

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

At this point all the basic functionalities have been implemented.
Time spent: 2-3 hours
Lines of code written/updated: 86

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

Features addition

some introduction

Todo creation feature

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

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

After generating a new unique id it creates a new Todo object called *newTodo* with the *completed* field set to *false* . Adding the new Todo to the TodoProvider's state *todos* list requires a bit of workaround. The state of a stateful widget is immutable. It can only be changed by the *onChangeFilter* method. Unfortunately, the method add for lists is of type void and do not return a new list but instead add the new value to the existing one. For this reason directly calling the add method to the TodoProvider's local lists *todos* will have no effect. That list is immutable and cannot be changed. TodoProvider's *todos* list must be completely replaced with a new list containing also the new todo. First a new temporary list called *newList*is created and populated with the element present in the *todos* list. Then the *newTodo* is added to this *newList*list. At this point is sufficient to replace the *todos* list with the new one inside the *onChangeFilter* method. To make this new function accessible down the tree is sufficient to add a new field in the TodoInheritedData (called onAddTodo) widget and pass the function on creation.

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

  {...}

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

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

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

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

```
class AddTodoPage extends StatefulWidget {

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

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

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

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

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

The current route (AddTodoPage) is also popped after the todo creation, and the HomePage is rebuilt (by the fact the TodoInheritedData changed).
Time spent: 20-30 minutes
Lines of code written/updated: 24
Classes/widget created: 0

Todo updating feature

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

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


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


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

this new function will be called *onUpdateTodo* and takes three arguments: the id of the todo to be updated, the *newName* that should be set and the *newDesc*. It first checks if a todo matching the id exists. Then, for the same immutability concept we dealt with when we spoke about the *onAddTodo*feature, a *newTodosList* is created and populated with the elements inside the *todos* list. Moreover, the todo with the corresponding id is update with the new name and new description. Finally, the *todos* list in the TodoProvider stateful widget is overridden with the *newTodosList* using the *onChangeFilter* method.

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

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

The next step is to wrap the Row widget in the *todo$_i$tem.dart* file inside a InkWell widget to make it sensible to taps.

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

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

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

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

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

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

A further change must be done in the MaterialApp's "/updateTodo" route to populate the field of the UpdateTodoPage correctly.

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

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

```
TextButton(onPressed: () {

  widget.callback(textControllerName.text,textControllerDesc.text);
  Navigator.pop(context);
},
```

Once pressed the UpdateTodoPage will be popped, and the HomePage rebuilt to show the actual changes in the *todos* list.
Time spent: 20-30 minutes
Lines of code written/updated: 43
Classes/widget created: 1 for arguments between routes

Render optimizations

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

```
class TodoInheritedData extends InheritedWidget {
```

to

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

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

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

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

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

In particular the TodoView widget will pass as aspect the number 0 declaring that should be notified (and rebuild) only when a *filteredTodos*'s structure change occurs. Instead TodoItem widgets will pass the corresponding Todo's id as aspect parameter. Now that every widget is registered only to the desired aspect of the data, is necessary to "teach" the TodoInheritedData to recognize which aspect of the data actually changed when a state change occurs. To do so InheritedModel provides a method called updateShouldNotiy-Depenedent that is just like the InheritedWidget's one *updateShouldNotify* but this time takes as argument also a Set of ints called *dependencies* (aspects). This method is called once for every widget that registered to state changes and the *dependencies* variable will contains all aspects the widgets registered to (only one for widget in our case). As follow the implementation of the method:

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

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

```
if( widgetRegisteredForStructureChange && strucutureChangeOccured){
        return true;
    }else{
        if( widgetRegisteredForSpecificTodoChange && thatTodoChanged){
            return true;
            }else{
                return false;
    }
}
```

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

```
class TodoItem extends StatelessWidget {
  final int id;

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

  @override
  Widget build(BuildContext context) {
    final Todo todo = TodoInheritedData.of(context, aspect: id)
        .todos
        .where((element) => element.id == id)
        .first;
```

At this point the application is working as intentioned and the renders optimization was successfully accomplished.
Time spent: 8-10 hours
Lines of code written/updated: 49
Classes/widget created: 0

2.2.3. Redux implementation

2.2.4. BloC implementation

2.2.5. MobX implementation

2.2.6. GetX implementation

3 | Conclusions and future developments

A final chapter containing the main conclusions of your research/study and possible future developments of your work have to be inserted in this chapter.

Bibliography

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 Symbols

Variable	**Description**	**SI unit**
$\boldsymbol{u}$	solid displacement	m
$\boldsymbol{u}_f$	fluid displacement	m

Acknowledgements

Here you might want to acknowledge someone.