Sommario

[Introduction 2](#_Toc88267235)

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

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

[1.2 Redux 3](#_Toc88267238)

[1.3 Bloc 3](#_Toc88267239)

[1.4 MobX 3](#_Toc88267240)

[1.5 GetX 3](#_Toc88267241)

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

[2.1 General overview 4](#_Toc88267243)

[2.1.1 Base functionalities 4](#_Toc88267244)

[2.1.2 Adding new features 5](#_Toc88267245)

[2.1.3 Renders optimization 5](#_Toc88267246)

[2.2 Implementation 6](#_Toc88267247)

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

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

[2.2.3 Redux 29](#_Toc88267250)

[2.2.4 BLoC 36](#_Toc88267251)

[2.2.5 MobX 57](#_Toc88267252)

[2.2.6 GetX 68](#_Toc88267253)

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

[4. Comparisons 69](#_Toc88267255)

[5. Conclusions 69](#_Toc88267256)

# The Todo app

This chapter describes the implementation of a mobile application handling a list of todos. The application is developed once for each of this four state management solutions:

* State objects + Observer components + context
* BLoC + Stream components + context
* Redux + Stream components + context
* MobX + context

The implementation of each application is divided in three sub-processes:

* Implementation of the base functionalities
* Features addition
* Renderings optimization

In the end, I propose a summary of the data collected during each implementation. Collected data refers to the lines of codes produced, and the time spent at each sub-process.

## General overview

This section describes in detail the output of each sub-process.

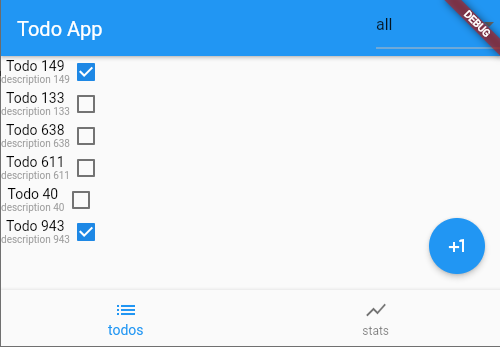
### Base functionalities

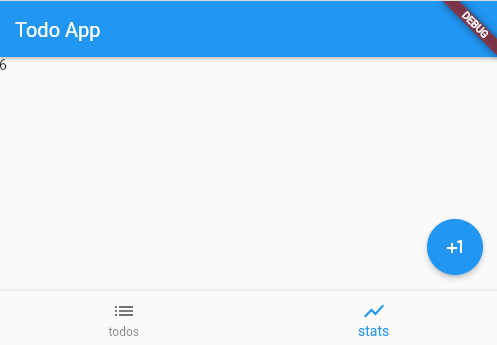
The output application of this sub-process is an application partially handling a list of todos.

It offers the possibility to:

* Visualize a list of todos with their names, description, and competition,
* Filter todos based on a filter (completed, pending or both),
* Visualize statistics about todos.

The output application is composed of a single page, called the HomePage. The HomePage content varies based on the value of a tab variable. When the tab is set to “todos” (see figure RIFERIMENTO) the HomePage shows the list of todos and provides a DropdownButton to filter them. When the tab is set to “stats” (see figure RFERIMENTO) the HomePage shows a numerical summary of the current todos situation.





### Adding new features

This step takes the output application of the previous step and add to its two new features.

The first feature introduces the possibility to add a new todo to the list. Another page, called AddTodoPage, is added to the application offering the possibility to fill up two input fields and to submit them to generate a new todo. Figure RIFERIMENTO show the UI of the AddTodoPage.

Immagine che contiene testo

Descrizione generata automaticamente

The second feature introduces the possibility to update an existing todo. Another page, called UpdateTodoPage, is added to the application. It can be reached from the HomePage tapping on a todo and offers the possibility to fill up two input fields to modify the todo. Figure RIFERIMENTO show the UI of the UpdateTodoPage.

Immagine che contiene testo

Descrizione generata automaticamente

### Renders optimization

This step aims at minimizing the number of widgets rebuild at every state change. It mostly targets the problem introduced here RIFERIMENTO, when a list with an arbitrary number of elements must be displayed. In short, without any optimization, the entire list of todos gets rebuilt when one of the elements changes, worsening performances and memory consumption. (large scenarios)

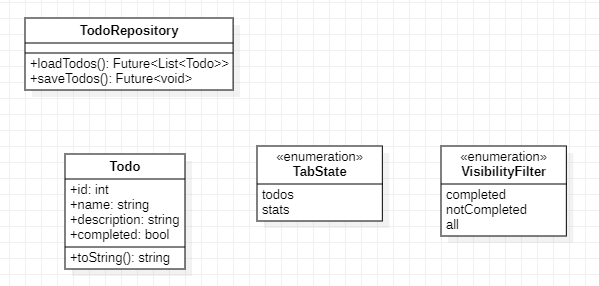
## Implementation

### Shared project structure and files

This subsection presents the parts of the code shared between different implementations.

**Models**

Let’s start defining models. Class diagram RIFERIMENTO shows the classes used to build the application state.

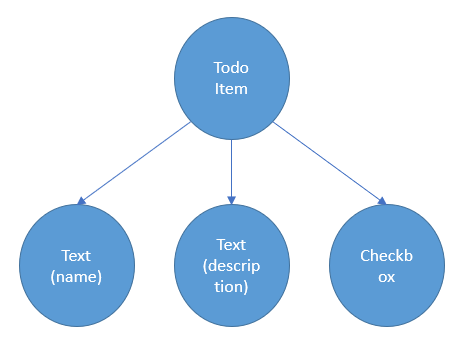


The TodoRepository class simulates the retrieval of a list of todos from a database. The loadTodos and saveTodos methods are asynchronous and produce a delay of 2 seconds to simulate the retrieval process. The loadTodos method return a list containing six todos with random ids.

**User Interfaces**

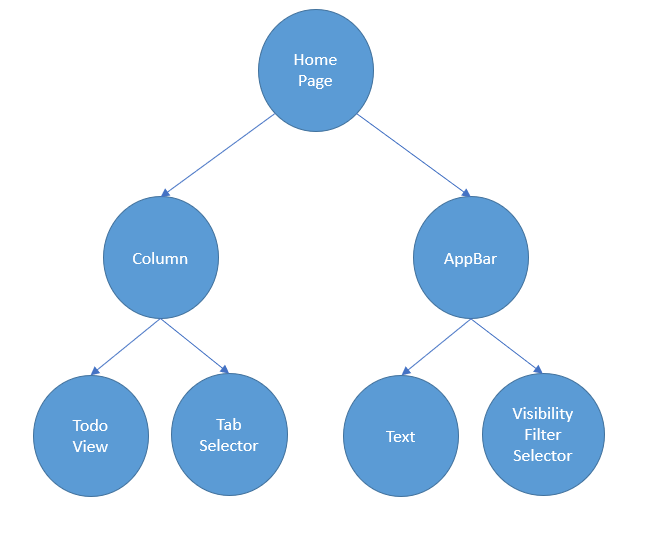
I start listing and describing the special purpose widgets used to build the three main pages. These widgets are:

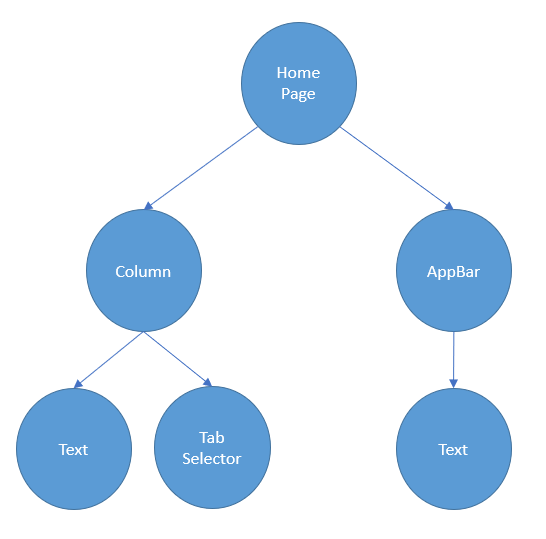
* The TodoItem widget. It displays information about a specific todo and enables changing its competition field. Figure RIFERIMENTO show the simplified widget tree structure of a TodoItem.



* The TodoView widget. It is a special purpose widget that takes a list of todo models and displays it. It converts the list to a ListView widget containing as many TodoItems as the length of the list.
* The VisibilityFilterSelector widget. It is a special purpose widget that enables swapping between filters using a DropdownButton.
* The TabSelector widget. It is a special purpose widget that enables swapping between tabs using a BottomNavigationBar.

Special purpose widgets are used to compose pages. Figure RIFERIMENTO shows the widgets tree of the HomePage when the tab value is set to “todos”, whereas figure RIFERIMENTO shows the widgets tree of the HomePage when the tab value is set to “stats”.





AddTodoPage and UpdateTodoPage share the same UI structure. They are both structured as figure RIFERIMENTO.

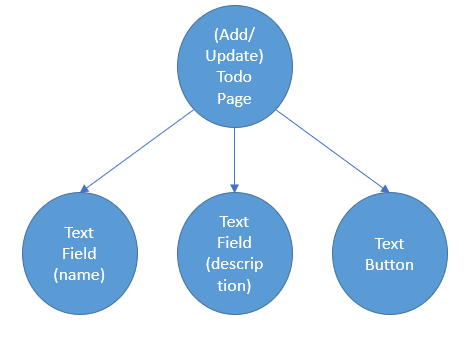


Table RIFERIMENTO shows a summary of the lines of code and classes used to implement the shared structure.

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

**Classes/widget created: 13**

### Inherited widget/model and SetState implementation

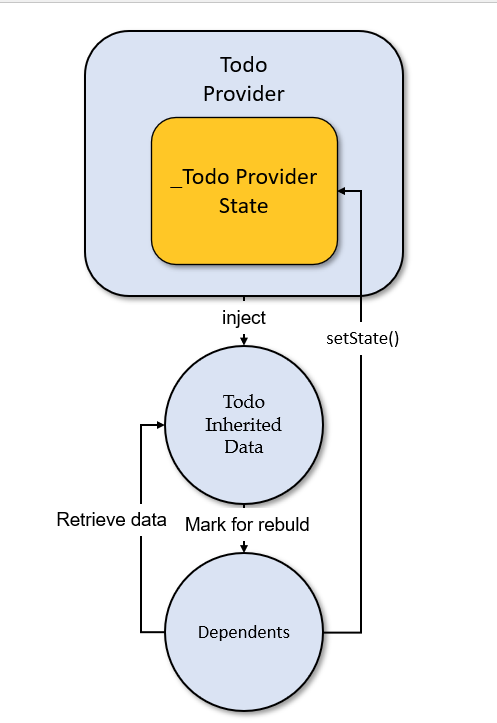
This section describes the implementation and the architecture of a todo application (see HERE RIFERIMENTO) using a complete state management solution. The questioned solution uses: state objects to contain state, context to dispatch it and observer components to kept UI synchronized. In particular, state dispatchment and dependents rebuilding are performed using InheritedWidgets.

As seen in their introduction, InheritedWidgets require the state to reside in a stateful widget. The stateful widget injects data in a InheritedWidget and makes it accessible in the subtree. In this implementation, the stateful widget is named TodoProvider, the InheritedWidget TodoInheritedData.

I treated the tab value as an ephemeral state and the rest of the application state as a shared state. In particular the shared state contains:

* The list of todos
* The filtered list of todos
* The current visibility filter
* The stats

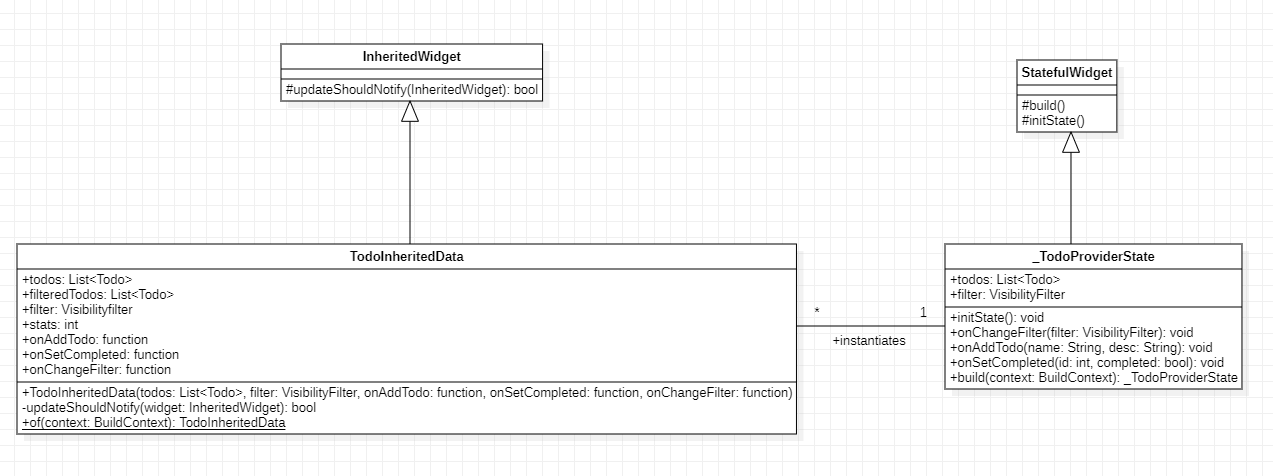
Let’s start visualizing the entire architecture with figure RIFERIMENTo.



This architecture has been explained here RIFERIMENTO. After the UI is built the first time, the workflow is the following:

1. A dependent (or the TodoProvider widget itself) updates the \_TodoProviderState with setState.
2. TodoInheritedData widget is rebuilt with the new state information(injected)
3. TodoInheritedData widget marks all listening widgets in the subtree for a rebuild
4. Dependents marked for a rebuild start the build process
5. Rebuilding dependents access the TodoInheritedData widget using the static of method to retrieve data they need
6. Dependents terminate the building process, and the UI is updated

Class diagram RIFERIMENTO shows a more accurate representation of the TodoProvider widget and the TodoInheritedData widget.



Notice that:

* \_TodoProviderState hold the actual list of todos and the current visibility filter
* onChangeFilter, onAddTodo and onSetCompleted are defined in the \_TodoProviderState and implicitly call setState method to update the list and the filter
* TodoInheritedData constructor receives a reference of the state changing functions from the \_TodoProviderState. Dependents can access the state changing functions directly through the TodoInheritedData instead of receiving them from the parent widget (props drilling)
* TodoInheritedData constructor just receives the list of todos and the filter. The filtered list and the stats are calculated later.
* The updateShouldNotify method compares the current list and the current filter with the old ones, in case they are both equal it returns false avoiding dependents to rebuild
* The \_TodoProviderState initState() method is the first to be executed and contains the actual fetching of the list from the fake repository

Here a summary of the collected data during the implementation of the base functionalities:

**Time spent: 2-3 hours**

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

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

The features addition process is pretty linear. It requires the following steps to be performed:

* Add two new functions in the TodoProvider stateful widget called onAddTodo and onUpdateTodo. They respectively add and update a todo in the list implicitly calling setState.
* Add two new parameters in the TodoInheritedData to contain the new functions references.
* Make the TodoProvider pass the reference of the new functions to the TodoInheritedData in the build method.
* Move the TodoProvider widget higher in the tree, above the MaterialApp widget. This step is necessary because, during the implementation of the base functionalities, the TodoProvider widget was located below the MaterialApp, just after the HomePage. However, when a new route is pushed it is inserted as a direct child of the MaterialApp, on the same level of the HomePage. For this reason, information contained in the TodoProvider would not be accessible by the AddTodoPage and by the UpdateTodoPage without lifting the TodoProvider above the MaterialApp.
* Retrieve the reference of the functions in the AddTodoPage and in the UpdateTodoPage and fire them on button press

Here a summary of the collected data during the addition process:

**Time spent: 20-30 minutes**

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

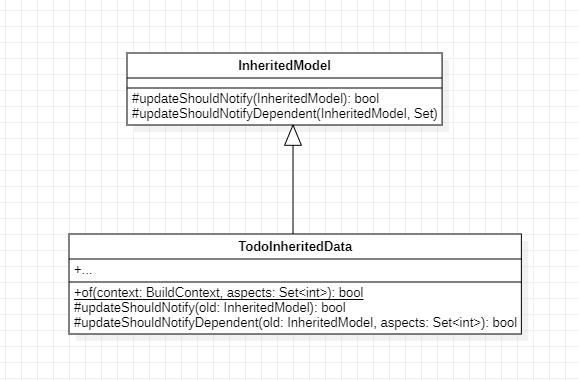
**Classes/widget created: 0**

Optimize renderings was a pretty hard task. I spent hours trying to make the single TodoItem rebuild instead of the entire TodoView before realizing that it was just unfeasible using plain InheritedWidgets. Searching for a solution, I came across InheritedModels that target this exact use case. InheritedModels are introduced here RIFERIMENTO.

An InheritedModels dependent listens for one or more aspects of the overall InheritedModel. It communicates the aspects it listens to when it calls the of static method exposed by the InheritedModel.

InheritedModels provide an additional protected method called updateShouldNotifyDependent that is run for every listening widget with the corresponding set of aspects it subscribed to. If the method evaluates to true, the widget rebuilds, elsewhere it does not.

Class diagram RIFERIMENTO shows the updated TodoInheritedData class.



To leverage this mechanism, I set up a mapping from int numbers to model aspects. I indicated with the number:

* 0: a change that affects the entire list of todos. For example, adding a todo or deleting a todo. This kind of change require the entire TodoView to rebuild. (no todo with id 0 for convention)
* N (where N is the id of the todo): a change that affect only the todo with id N.

At this point, I made the TodoView access the InheritedModel passing the number 0 in the aspect parameter, whereas the single TodoItem access the InheritedModel passing the id of its own todo.

The last thing to do is to override the updateShouldNotifyDependent function implementing the logic though which changes in the model are bound to aspects.

This function was pretty hard to code, I report just the pseudocode for simplicity (see Source code RIF).

@override  
bool updateShouldNotifyDependent(

if(changeAffectingTheEntireListOccured){

// this leads every dependent to rebuild whatever aspect it subscribed to

return true;

}else{

// in case the change is not affecting the entire TodoView

//check which aspect the dependent subscribed to

if(dependencies.contains(0)){

//if it subscribed to structural changes

//do not rebuild because the change is not structural

return false;

}

If(todoWithID(Dependencies).changed)

// in this case, the dependencies (Set) variable contains the Id of the todo the dependent subscribed to

//if the todo with id equal to the value in the dependencies changed

// evaluate to true (rebuild)

return true;

}

}

//if no previous statements were satisfied return false

return false;

}

Here a summary of the collected data during the optimization process:

**Time spent: 8-10 hours**

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

**Classes/widget created: 0**

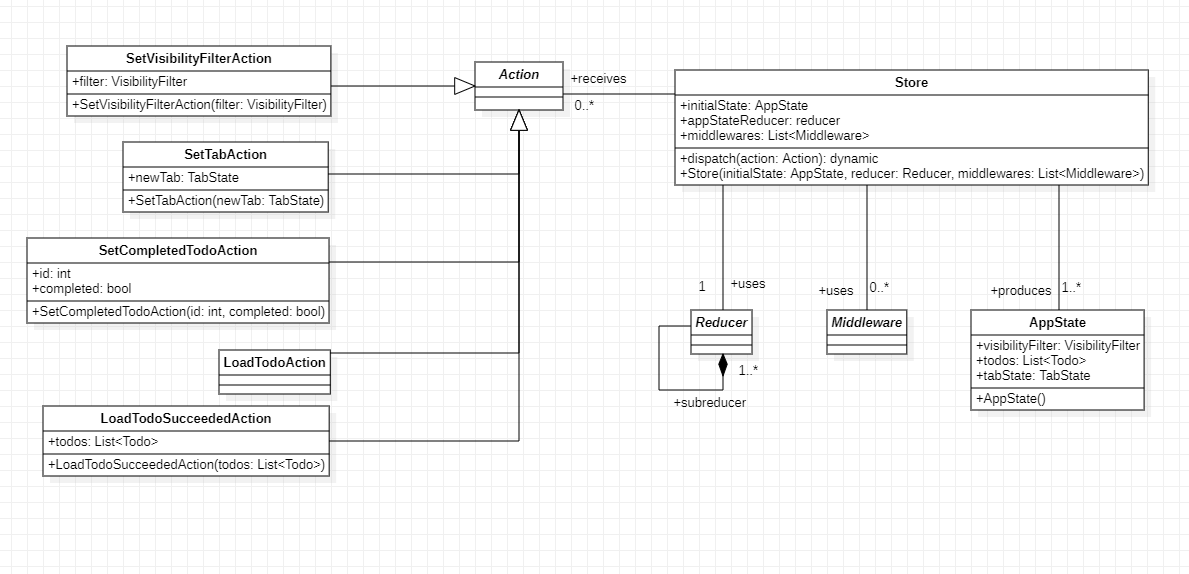
### Redux implementation

This section describes the implementation and the architecture of a todo application (see HERE RIFERIMENTO) using a complete state management solution. The questioned solution uses: Redux pattern to handle state, context to dispatch it and stream components to kept UI synchronized. The implementation uses two Flutter packages: redux(version) and flutter\_redux(version) (see RIFERIMENTO).

I treated the entire application state as a shared state. In particular, the shared part contains:

* The list of todos
* The current visibility filter
* The current tab value

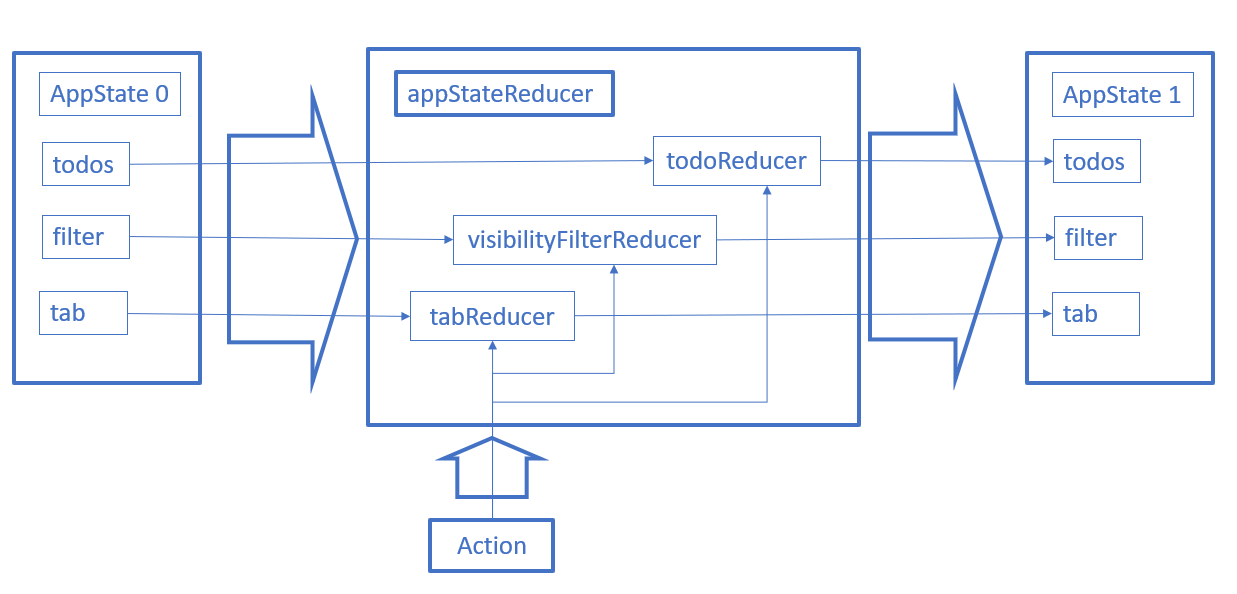
Let’s start defining the elements of the Redux store. Class diagram Rif shows the model of the application state plus the models of the actions.



Notice:

* LoadTodoAction starts the fetching process
* LoadTodoSucceededAction ends the fetching process and contains the fetched list
* The Store class is provided by the *redux* package and exposes a dispatch function taking a generic action as single argument
* I omitted other general-purpose functions from the store class definition for simplicity
* The constructor of the AppState does not take any argument; the list is set to be empty, the filter is set to “all” and the tab is set to “todos” by default

Let’s go ahead with a picture representing the appStateReducer and its internal composition (see figure RIFERIMENTO).



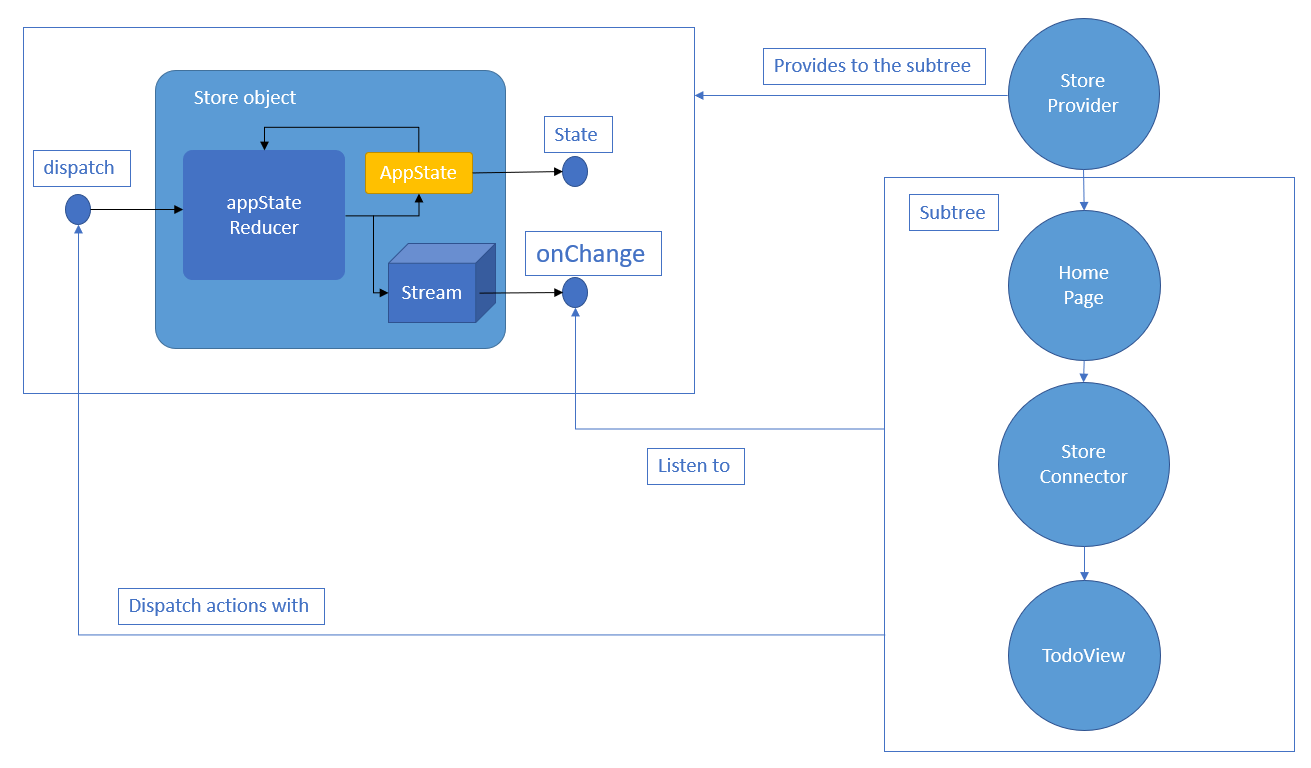
The appStateReducer takes an AppState and an Action and returns a new AppState. The appStateReducer is composed of three subreducers, each handling a specific aspect of the AppState. The output AppState is filled with the output values of each subreducer.

Actions go through each subreducer. If the action does not match any of the actions handled by the subreducer, the subreducer returns the current value of the state.

The application uses a single middleware, called loadTodosMiddleware. Its implementation is shown in Source code riferimento.

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

The middleware is used to handle the fetching process. A middleware is just a void function taking three parameters: the store, the action and the next middleware. The loadTodosMiddleware middleware intercepts actions before they reach the appStateReducer. The NextDispatcher is the next middleware in the list or, if no other middleware is present, the reducer. The loadTodosMiddleware reacts when the dispatched action is of type LoadTodoAction. Before passing the action to the appStateRecuder it starts the fetching process. It also takes care to dispatch an action of type LoadTodoSucceededAction when the fetching process ends.

Let’s go ahead presenting the general architecture of the application with figure RIFERIMENTO. 

The application uses a StoreProvider widget situated on top of the widget tree. Several StoreConnector widgets retrieve the Store using the of method and listen to states coming out of the *onChange* stream. Widgets that intend to mutate the AppState call the *dispatch* method with an action as payload.

Widgets listening to the Store with their corresponding viewmodels are:

* The HomePage which depends on the tab value (TabState)
* The TodoView which depends on the list and on the filter (List<Todo>)
* The Stat which depends on the list of todo (int)
* The TabSelector which depends on the tab value (TabState)
* The VisibilityFilterSelector which depends on the filter value (VisibilityFilter)

Widgets changing the state are:

* The TabSelector which mutates the tab value dispatching a SetTabAction
* The VisibilityFilterSelector which mutates the filter value dispatching a SetVisibilityFilterAction
* Every TodoItem which mutates the list of todos dispatching a SetCompletedTodoAction
* The HomePage which starts the fetching process inside the initState method dispatching a LoadTodoAction

Notice that the AppState does not contain the filtered list nor the stats. They are computed in the visualization layer respectively by the TodoView and Stats StoreConnectors. Moreover, the TodoView is rebuild every time a single todo changes because its viewmodel changes with it. This implies that the filtered list is recomputed every time a checkbox is tapped.

Understanding the Redux pattern was tough, but the implementation of the base functionalities was pretty linear. Most of the boilerplate came out from the action definitions and the StoreConnector widgets.

Here a summary of the collected data during the implementation of the base functionalities:

The features addition process required the following steps:

* Add two new actions called AddTodoAction and UpdateTodoAction
* Update the todosReducer to handle the new actions
* Retrieve the Store and dispatch an AddTodoAction in the TextButton of the AddTodoPage
* Retrieve the Store and dispatch an UpdateTodoAction in the TextButton of the UpdateTodoPage

Here a summary of the collected data during the addition process:

**Time spent: 20-30 minutes**

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

**Classes/widget created: 0**

The optimization process leverages the fact that StoreConnectors rebuild when their viewmodel changes.

These are the steps necessary to optimize the TodoView renderings:

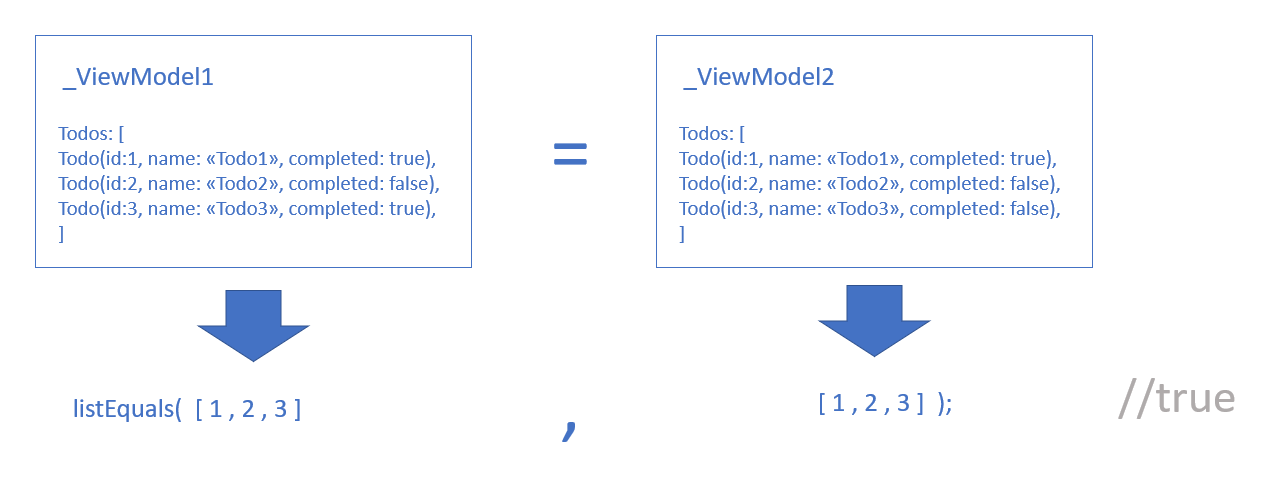
* Make the TodoView viewmodel differs from the previous one only when a structural change occurs
* Wrap every TodoItem into a StoreConnector to rebuild them independently from the TodoView

Let’s start with the first step. A TodoView widget should change when a structural change occurs meaning that a todo is added or remove from the list or both cases together. Currently, the TodoView rebuilds at every state change because its viewmodel (a List<Todo>) always differs from the previous one due to how Dart compares objects. In fact, comparing two identical lists (containing the same values) evaluates to false because they are different instances. To change this behavior, we can wrap the list into a new class and override its == operator.

The new class is called \_ViewModel and is private of the TodoView. Source code RIF reports its implementation.

class \_ViewModel {  
 final List<Todo> todos;  
  
 \_ViewModel({required this.todos});  
  
@override  
 bool operator ==(Object other) {  
 if(other is \_ViewModel) {  
 List<int> ids = todos.map((todo) => todo.id).toList();  
 List<int> otherIds = other.todos.map((todo) => todo.id).toList();  
  
 return listEquals(ids,otherIds);  
 }else{  
 return false;  
 }  
 }  
}

The == operator simply maps both filtered lists to new ones containing the ids only before applying the *listEquals* function. Figure Rif shows the behaviour of the == operator when a non-structural change occurs (the change marks the Todo3 as pending). The comparison between \\_ViewModel1 and \\_ViewModel2 evaluates to true leading the TodoView as it is.



Let’s proceed with the second step and wrap the TodoItem widget inside a StoreConnector. The converter function takes the store and selects the todo with the corresponding id. The id is passed to the TodoItem at its creation. Source code Rif shows the converter function of a TodoItem.

(store) =>store.state.todos.firstWhere((element) => element.id == id),

The equality operator for Todo instances checks recursively if all fields match and if the other object if of type Todo. This leads a TodoItem to rebuild when one field of the todo it is listening to changes. In case exposed by figure RIF, Todo1 and Todo2 do not rebuild because their viewmodel is not changed, Todo3 does.

The optimization process was easy.

Here a summary of the collected data during the addition process:

**Time spent: 20-30 minutes**

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

**Classes/widget created: 0**

### Implementazione con BLoC

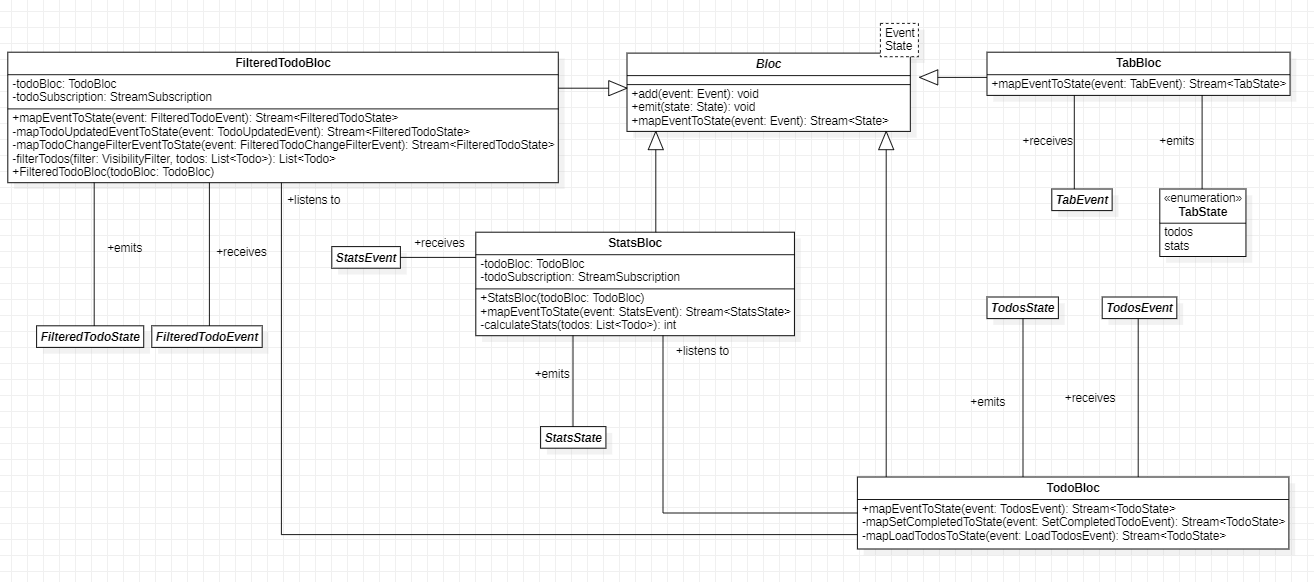
This section describes the implementation and the architecture of a todo application (see HERE RIFERIMENTO) using a complete state management solution. The questioned solution uses: the BLoC pattern to handle state, context to dispatch it and stream components to kept UI synchronized. The implementation uses two Flutter packages: bloc(version) and flutter\_bloc(version) (see RIFERIMENTO).

The whole application state is treated as a shared state. It is divided into four blocs:

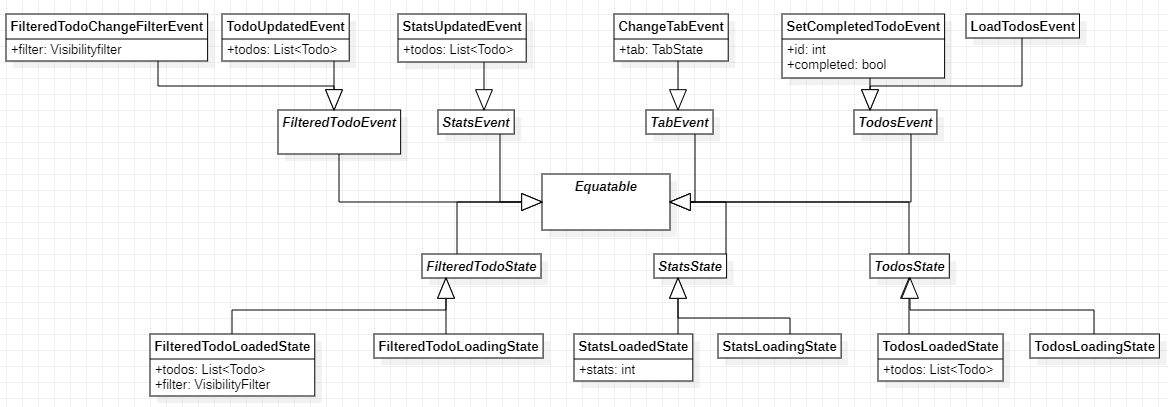
* The TodoBloc that handles the state of the list of todos
* The StatsBloc that handles the state of the stats. This bloc reacts to the TodoBloc stream of state
* The FilteredTodoBloc that handles the state of the filtered list. This bloc reacts to changes in the TodoBloc
* The TabBloc that handles the state of the tab

Class diagram in figure RIF shows the structure of the application state. The four blocs extend the Bloc class (from the bloc package) which provides the *add* method and the *emit* method. The former is used to receive events, the latter is used to emit states in the output stream. The Bloc class also requires the *mapEventToState* method to be overridden for each inheriting class. The FilteredTodoBloc and the StatsBloc contains a StreamSubscription variable to listen to the TodoBloc.

(Notice that every bloc receives abstract events and emits abstract states.)



Class diagram in figure RIF shows the hierarchy of events and states. Both events and states extend the abstract Equatable class that automatically overrides the equality operator, removing a great amount of boilerplate.



Notice that a BLoC state can be intended as a collection of objects each containing a specific part of the state, whereas a Redux state is a single object of type AppState.

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

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

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

**Lo stato dei todo**

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

\mbox{}

\end{code}

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

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

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

**The state of the stats**

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

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

**The state of the tab**

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

enum TabState{  
 todos,stats  
}

**Events**

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

**Events of the list of todos**

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

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

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

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

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

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

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

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

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

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

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

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

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

**The Blocs**

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

**The bloc for the list of todos**

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

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

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

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

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

}

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

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

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

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

}13

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

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

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

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

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

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

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

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

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

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

**The bloc for the stats**

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

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

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

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

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

**The bloc for the tab**

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

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

**Observe blocs**

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

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

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

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

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

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

**Making the state accessible**

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

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

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

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

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

**State injection in the UI**

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

**The HomePage**

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

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

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

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

**The TodoView Component**

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

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

11

**The TodoItem component**

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

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

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

**The VisibilityFilterSelector component**

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

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

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

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

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

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

**The TabSelector component**

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

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

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

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

3

**The Stats component**

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

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

6

**Conclusions**

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

In the follow some summarizing data re reported:

Time spent: 10-12 hours

Lines of code written/updated: 367

Classes/widget created: 24

Created files: 12

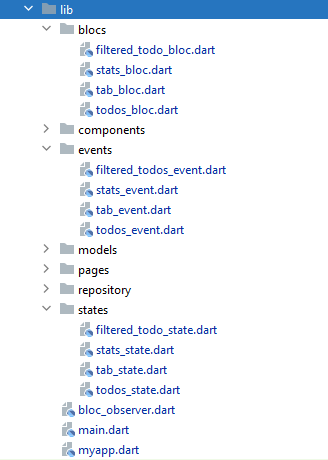
Tutte le funzionalità base dell’applicazione sono state sviluppate e funzionano correttamente. In generale il processo di sviluppo non ha avuto grossi intoppi. La l’unica parte complessa è stato creare i due blocchi del filtro e delle statistiche. Essendo essi dipendenti da un blocco esterno hanno richiesto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

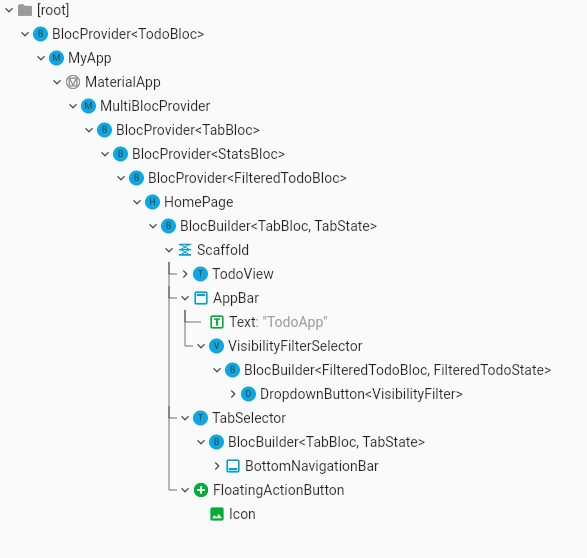
Tempo impiegato (per comprendere e scrivere la soluzione) : 10-12 ore

Linee di codice aggiunte/modificate: 367

Classi create: 24

File creati:12

 da inserire dopo la creazione del blocobserver



Da inserire nelle conclusioni

Immagine che contiene tavolo

Descrizione generata automaticamenteda inserire nelle ottimizzazioni

#### Features addition

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

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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

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

\end{minted}

\mbox{}

\end{code}

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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

Conclusions

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

**Time spent: 15-20 minutes**

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

**Classes/widget created: 2**

#### Rendering optimizations

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

@override  
Widget build(BuildContext context) {

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

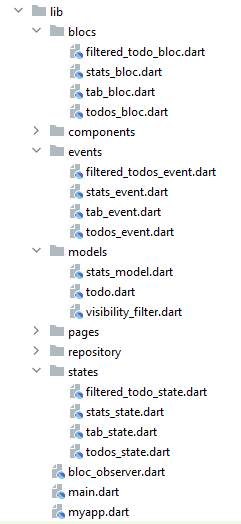
**Conclusions**

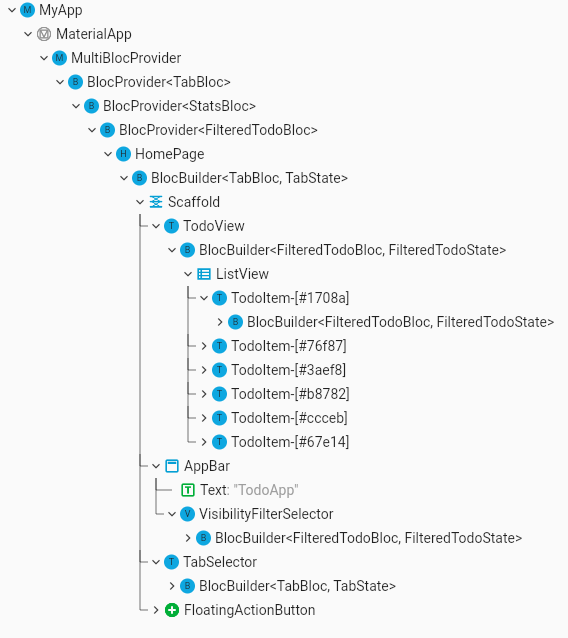
It was not an easy task to perform these optimizations.

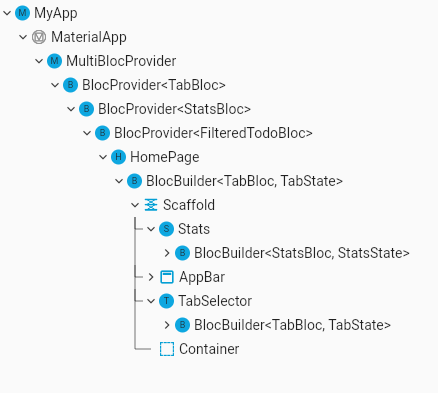
**Time spent: 6-8 hours**

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

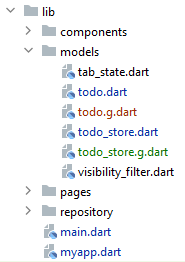
**Classes/widget created: 0**

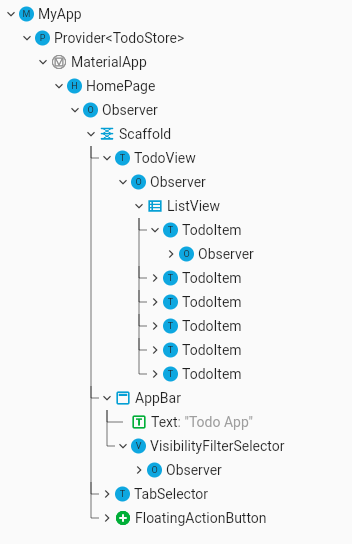


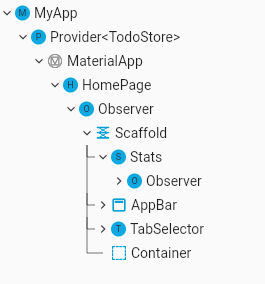




### MobX implementation

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





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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

\end{minted}

\mbox{}

\end{code}

**\paragraph**{Actions - }

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

\end{minted}

\mbox{}

\end{code}

**\paragraph**{Computed fields - }

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

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

\end{minted}

\mbox{}

\end{code}

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

\end{minted}

\mbox{}

\end{code}

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

Set the filter value to the tapped one.

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

The base functionalities are now working fine.

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

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

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

(…)

}

\end{minted}

\mbox{}

\end{code}

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

\begin{code}

\mbox{}\\

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

**\label{code:2.14}**

\begin{minted}{dart}

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

\end{minted}

\mbox{}

\end{code}

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

**The @readonly “issue”**

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

# The other app

# Comparisons

# Conclusions