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

Each implementation is divided in three sub-processes:

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

At the end of each implementation, I propose a summary of the data collected during the process. 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.

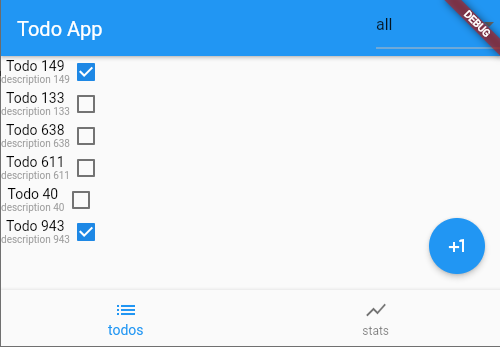
### Base functionalities

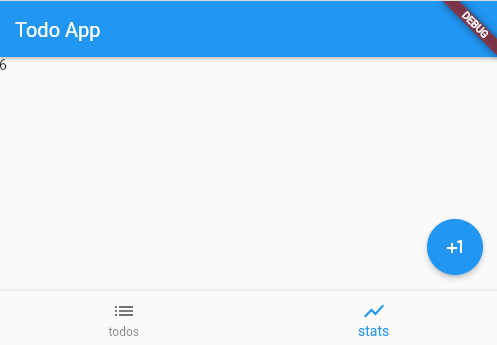
The output application of this sub-process is an application partially handling a list of todos.

It offers the possibility to:

* Visualize a list of todos with their names, descriptions, and competitions,
* Filter the list 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 adds to it 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 shows 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 shows 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 is displayed. In short, without any optimization, the entire list of todos rebuilds when one of the elements changes, worsening performances and memory consumption. (large scenarios)

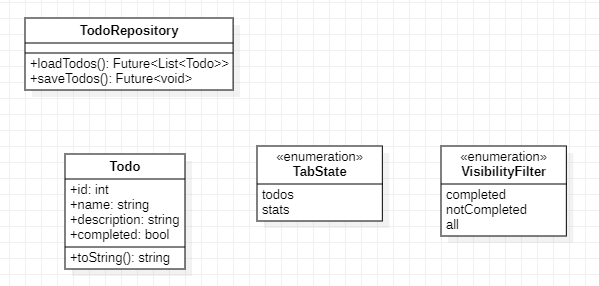
## Implementation

### 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.

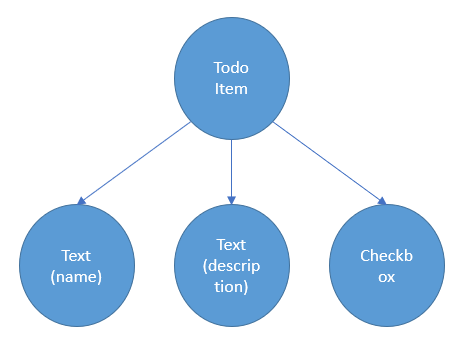


The TodoRepository class simulates the retrieval of a list of todos from a database. The loadTodos and saveTodos methods are asynchronous and produce a delay of 2 seconds to simulate the retrieval process. The loadTodos method returns a list containing six todos with random ids.

**User Interfaces**

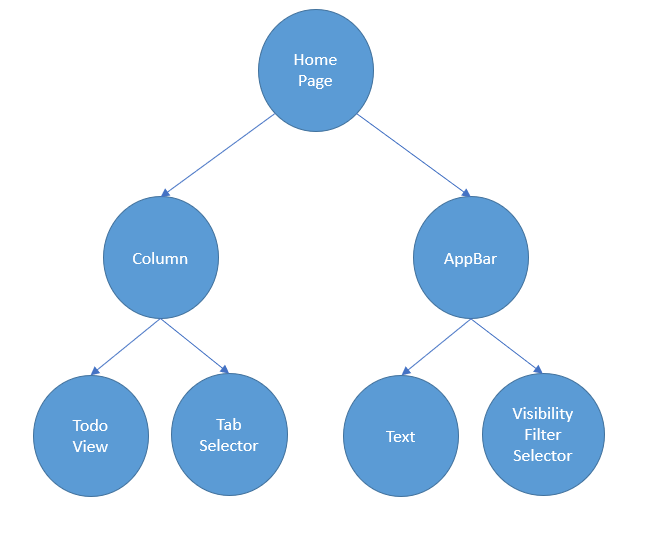
The special purpose widgets used to build the three main pages are:

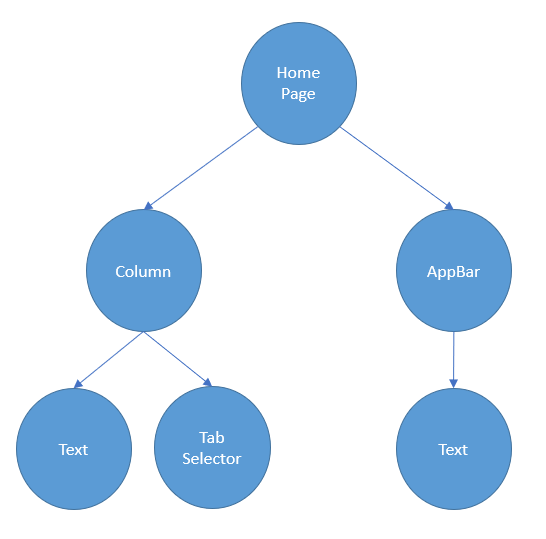
* The TodoItem widget. It displays information about a specific todo and enables changing its competition field. Figure RIFERIMENTO show the simplified widget tree structure of a TodoItem.



* 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.

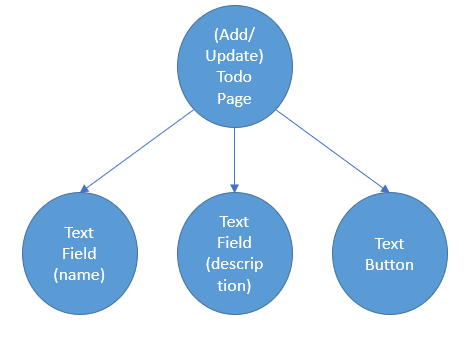


Table RIFERIMENTO shows a summary of the lines of code and classes used to implement the shared structure.

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

**Classes/widget created: 13**

### Inherited widget/model and SetState implementation

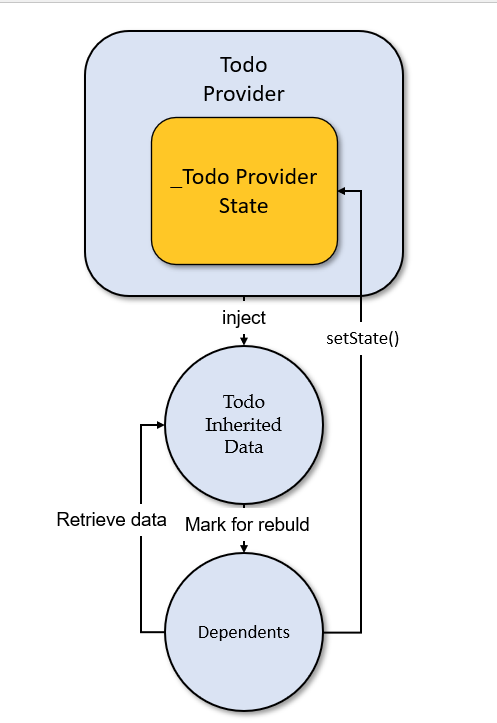
This section describes the implementation and the architecture of a todo application (see HERE RIFERIMENTO) using a complete state management solution. The questioned solution uses: state objects to contain state, context to dispatch it and observer components to kept UI synchronized. In particular,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

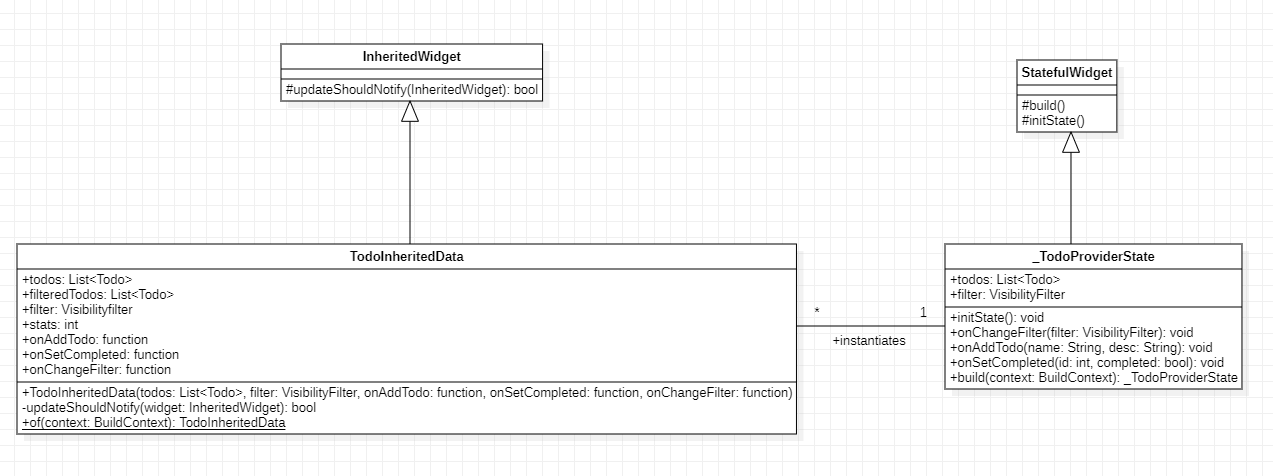
Let’s start visualizing the entire architecture with figure RIFERIMENTo.



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**

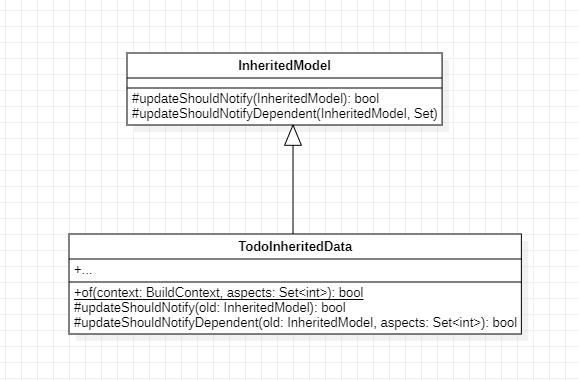
**Classes/widget created: 0**

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**

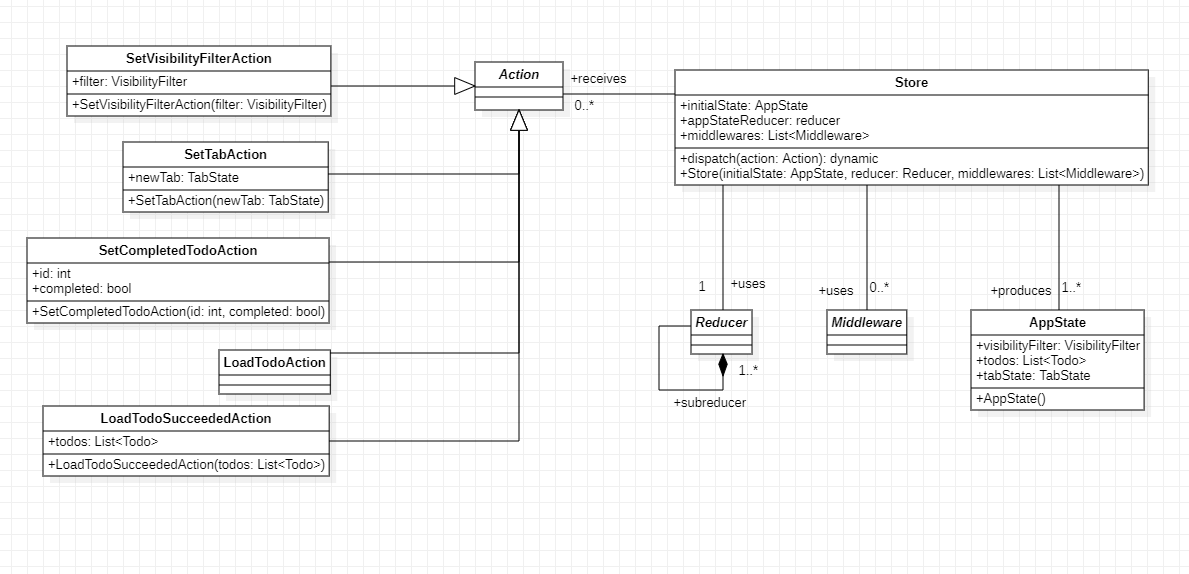
### Implementation with Redux

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 state contains:

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

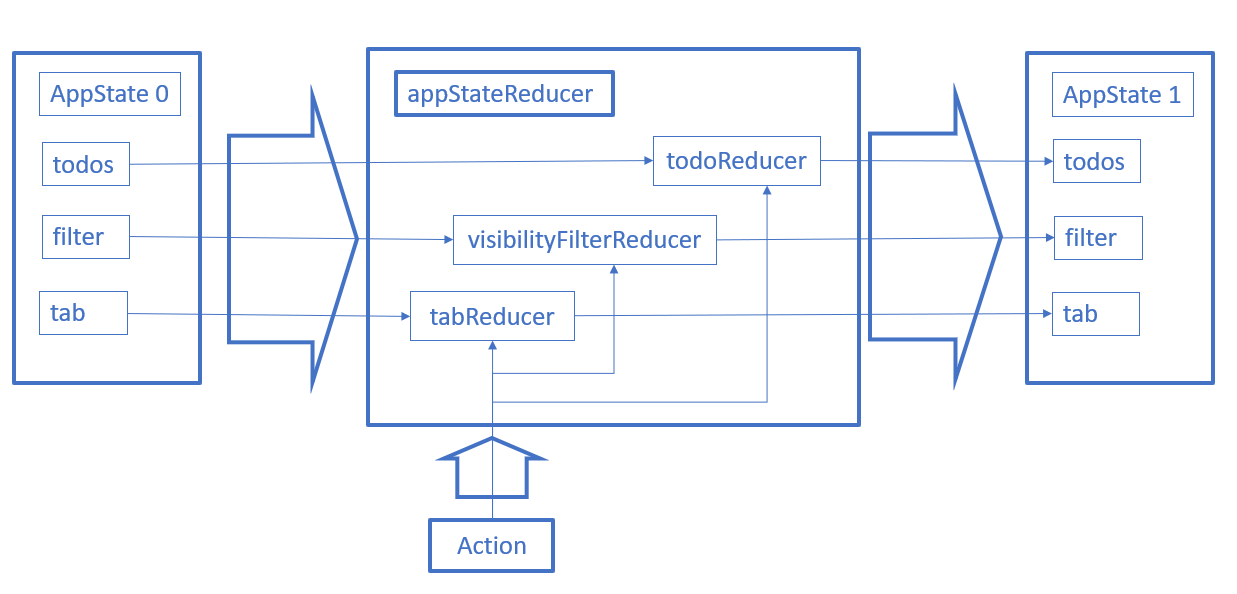
Let’s start defining the elements of the Redux store. Class diagram Rif shows the model of the application state plus the models of the actions.



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.

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 requires 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

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

These are the steps necessary to optimize the TodoView renderings:

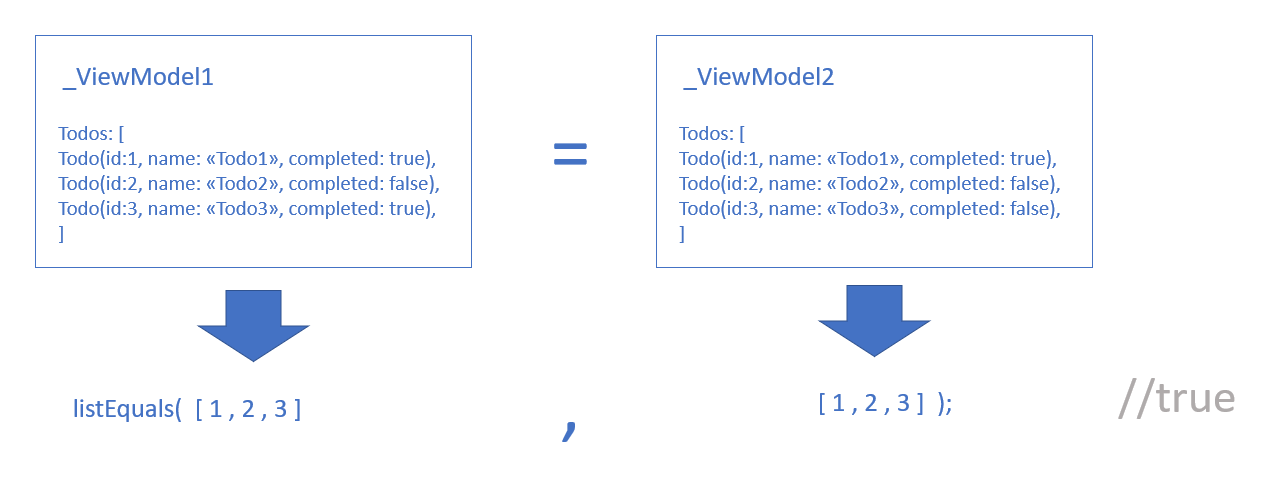
* Make the TodoView viewmodel differs from the previous one only when a structural change occurs
* Wrap every TodoItem into a StoreConnector to rebuild them independently from the TodoView

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).

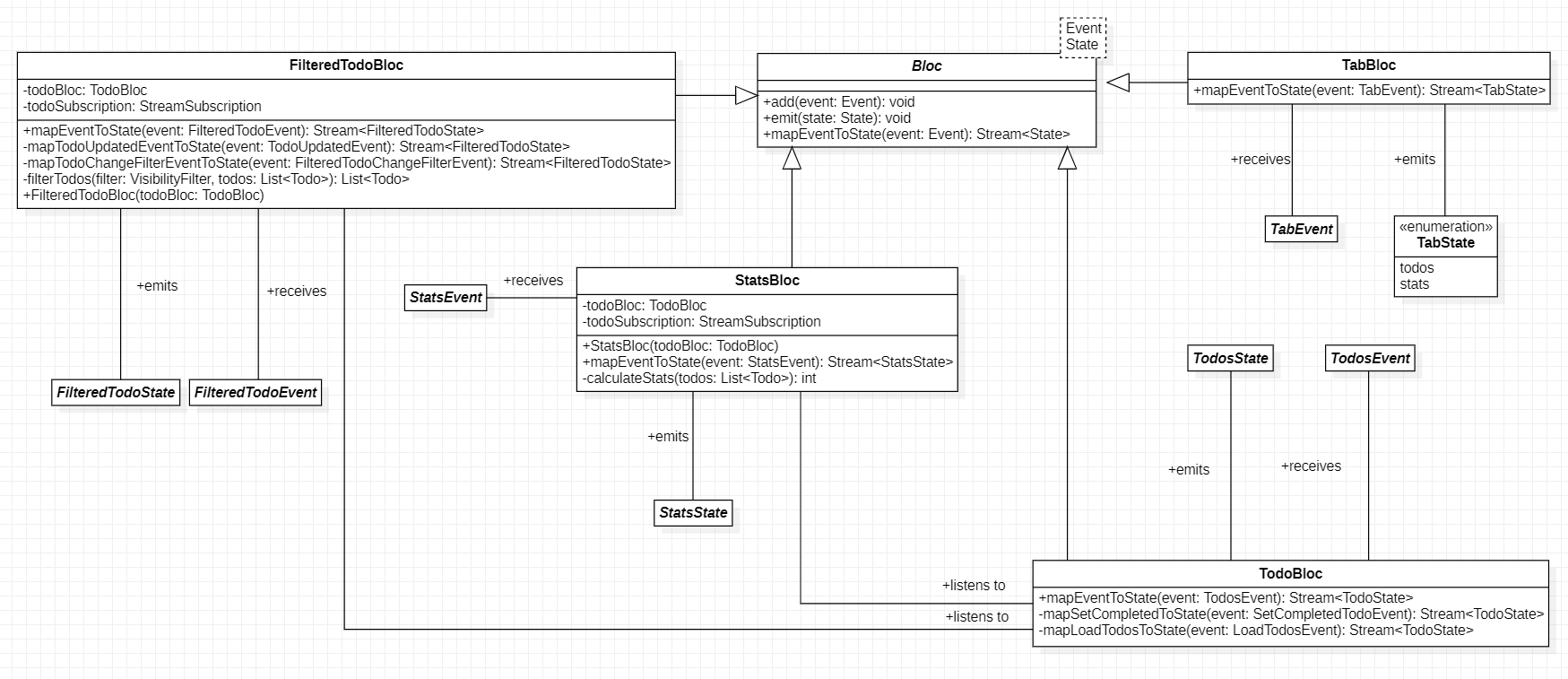
I treated the entire application state as a shared state. In particular, the shared state contains:

* The list of todos,
* The list of filtered todos,
* The current visibility filter,
* The current tab value.

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

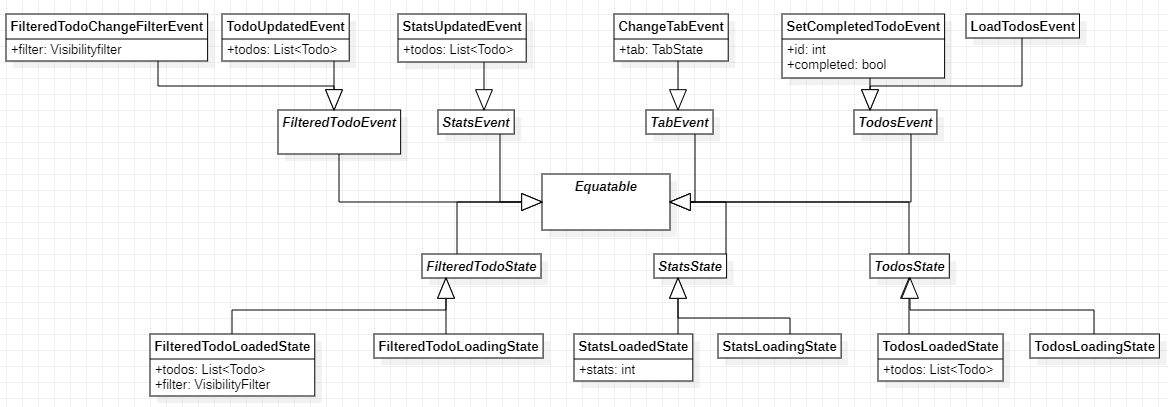
Class diagram in figure RIF shows the application blocs and their corresponding states and events.



Important concepts in the diagram:

* 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 listen to the TodoBloc using a StreamSubscription.
* The fetching of the list of *todos* from the database is performed in the TodoBloc constructor when it is instantiated.
* Every bloc receives abstract events and emits abstract states. This means that a bloc is able to produce and receive only a predefined set of states and events.

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



Note: an application state using the BLoC pattern is a collection of objects each containing a specific part of the state, whereas an application state using the Redux pattern is a single object of type AppState.

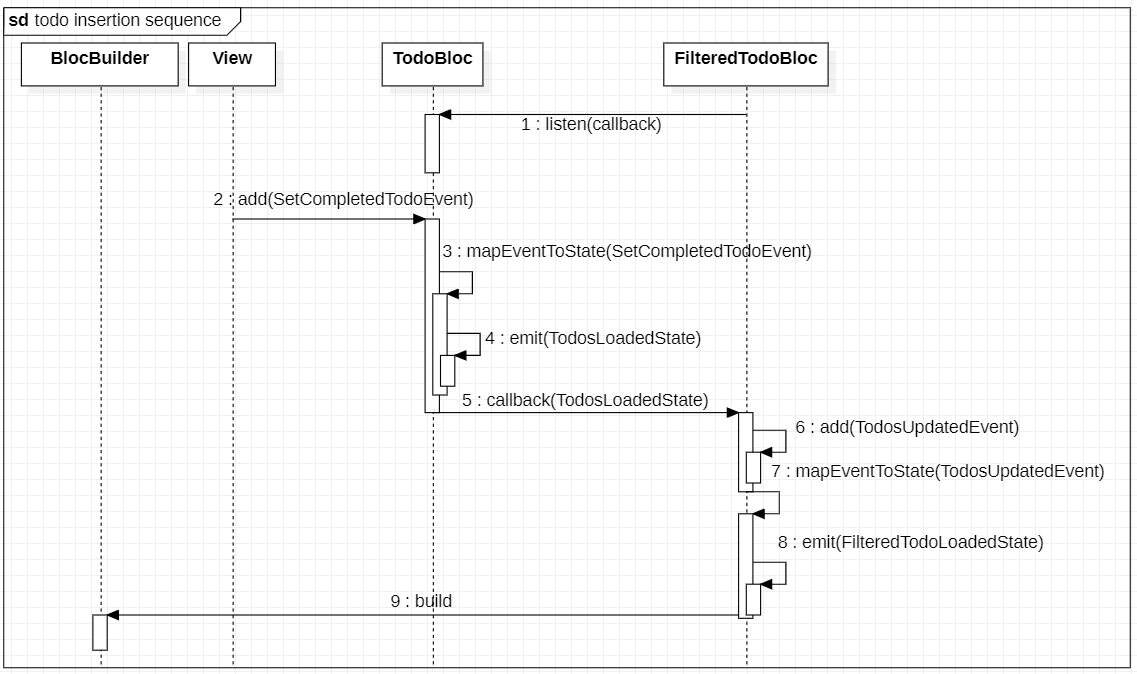
The application uses a single MultiBlocProvider at its root and multiple BlocBuilder widgets. Here a list of the widgets accessing the shared state with their corresponding watched bloc:

* The HomePage which listens to the TabBloc
* The TodoView which listens to the FilteredTodoBloc
* The Stat which listens to the StatsBloc
* The TabSelector which listens to the TabBloc
* The VisibilityFilterSelector which listens to the FilteredTodoBloc

Widgets changing the state are:

* The TabSelector which adds events in the TabBloc
* The VisibilityFilterSelector which adds events in the FilteredTodoBloc
* Every TodoItem which adds events in the TodoBloc

Now I would propose a sequence diagram to visualize the process of changing the competition of a todo. Figure RIFGERIMENTo describes the interactions between the logic layer and a generic BlocBuilder listening to the FilteredTodoBloc outgoing states.



1. The FilteredTodoBloc listen to the stream of states of the TodoBloc providing a callback function to be called when a new state pops out.
2. A generic widget in the View adds a SetCompletedTodoEvent in the TodoBloc ingoing stream (for example a CheckBox tapped in a TodoItem).
3. The TodoBloc consumes the event calling the *mapEventToState* method.
4. During the *mapEventToState* execution, a new state of type TodosLoadedState is emitted in the outgoing stream. (The new state is visible by all the listener of the stream)
5. The FilteredTodoBloc reacts to the new state emitted by the TodoBloc executing the callback function of step 1.
6. During the callback execution, a new event of type TodosUpdatedEvent is added to the FilteredTodoBloc event stream.
7. The FilteredTodoBloc consumes the event calling the *mapEventToState* method.
8. During the *mapEventToState* execution, a new state of type FilteredTodosLoadedState is emitted in the outgoing stream.
9. The BlocBuilder listening to the FilteredTodosBloc rebuilds due to the emission of the new state.

Note that, the callback function used to react to changes of the TodoBloc **does not emits any new state, it adds a new event in the FilteredTodosBloc bloc.** The emission of the new state (if any) is left to the *mapEventToState* method. This workflow has two advantages:

* This way, the entire logic that maps events to states is contained in a single method making much easier to understand the cause of a strange transition
* The emission of a new state is always preceded by an event receipt. Theoretically, the callback function could incorporate its own logic and produce a new state without emitting any event, however, this would violate the action-based mutations policy of the BLoC pattern.

The features addition process requires the following steps:

* Add two new events called AddTodoEven and UpdateTodoEvent (with their logic)
* Update the *mapEventToState* method of the TodoBloc to handle the new actions
* Retrieve the TodoBloc and dispatch an AddTodoEvent in the TextButton of the AddTodoPage
* Retrieve the TodoBloc and dispatch an UpdateTodoEvent in the TextButton of the UpdateTodoPage

The optimization uses a specific field of the BlocBuilder widget called *buildWhen.* It takes a function that compares the current state and the previous one and returns a Boolean value on which basis the BlocBuilder rebuilds.

These are the steps necessary to optimize the TodoView renderings:

* Provide the TodoView buildWhen field with a function containing the rebuilding logic
* Wrap every TodoItem into a BlocBuilder, this way they can rebuild independently by the TodoView

Overall, the procedure is similar to the one used with Redux (RIFERIMENTO) but presents an additional issue. Using Redux, the application state is “fixed”; the structure (is always the same)remains the same, whereas the data it contains changes over time. For example, an AppState always contains a list, a visibility filter and a tab and the view is built based on the values each variable takes. When the application starts, the AppState already contains a list, the list is empty meaning that it has not been fetched yet.

With BLoC, the application state is composed of a series of objects each representing a specific aspect of the state. The structure of the application state changes over time with the data is contains. For example, the part of the state concerning the list can be represented with an object of type TodosLoadedState or an object of type TodosLoadingState. When the application starts, the state is of type TodosLoadingState and does not contain a list at all. This means that, with bloc, it is not enough to check if the list changes in a transition but also that it actually exists.

Source code RIFEIRMENTO shows the implementation of the buildWhen function of the TodoView.

buildWhen: (previous, next) {

//if one of the states is of type Loading => rebuild  
 return !((previous is FilteredTodoLoadedState) &&  
 (next is FilteredTodoLoadedState) &&

//else check for structural changes  
 checkStructuralChange(previous.todos, next.todos));  
}

Source code RIFEIRMENTO shows the implementation of the buildWhen function of the TodoItem.

buildWhen: (previous, next) {

//if one of the states is of type Loading => rebuild  
 return (next is TodosLoadedState &&  
 previous is TodosLoadedState &&

//else check for changes in the todo

todoIsChanged(previous.todos, next.todos, id));  
}

### Implementation with MobX

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 five Flutter packages: mobx (2.0.5), flutter\_mobx (2.0.2),provider (6.0.1), build\_runner(2.1.4) and mobx\_codegen(2.0.4) (see RIFERIMENTO).

I treated the entire application state as a shared state except for the tab state. In particular, the shared state contains:

* The list of todos,
* The list of filtered todos,
* The current visibility filter.

I treated the current tab value as an ephemeral state. It is contained in the HomePage and is handled with the *mobx* package using an Observable variable.

Class diagram RIFERIMENTO shows the abstract class TodoStore used to generate the code in the todo\_store.g.dart file.

Immagine che contiene tavolo

Descrizione generata automaticamente

The application uses a Provider widget to dispatch the TodoStore to the sub-tree. Several Observer widgets in the sub-tree react to changes in the Store. Here a list of the position of the Observer widgets with their observed variable:

* One in the HomePage that observes the tab variable (local to the HomePage)
* One in the Stats that observes the stats variable
* One in the TodoView that observes the filteredTodos variable
* One in the VisibilityFilterSelector that observes the filter variable

Here the list of the widgets that update the Store:

* The Provider widget that calls the fetchTodos action at its creation
* The VisibilityFilterSelector that calls the changeFilter action
* The TodoItem that calls the setCompleted action

The feature addition process requires the following steps:

* Add two new actions in the TodoStore abstract class called addTodo and updateTodo with their logic
* Generate the code again
* Retrieve the TodoStore and call the addTodo action in the TextButton of the AddTodoPage
* Retrieve the TodoBloc and call the updateTodo action in the TextButton of the UpdateTodoPage

The optimization process was fast and easy as anticipated/foreseen in chapter RIFERIMENTO. It required two steps:

* Make the Todo model observable as well (as the list)
* Wrap every TodoItem in an Observer widget

The first step basically required to mark every variable in the Todo model as observable and to generate the corresponding code. After step 2, the MobX reactivity system was able to distinguish changes in the list and changes in a single todo.

This was the simplest optimization process and **did not require any logic definition**. Even if the other optimization processes were not difficult, they introduced another potential source of errors and performance issues that could be automatized and/or removed, as the MobX package does.

I would like to point out that implementation with MobX is the only one that required to modify the model definitions (in the optimization part). As stated HERE, the usage of the MobX package make the application even more dependent from the external packages.

Here a sfgnsdnndfnfenkfenknk

# The other app

# Comparisons

# Conclusions