**Introduction**

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

**The Todo app**

The first app we present the development of is a Todo app.

**Base functionalities**

It offers the possibility to visualize and partially handle todos. It is composed of a single page: the HomePage. The HomePage is composed by an appbar and two tabs: the todo tab and the stats tab.

In the todo tab the list of todos is visualized. Is possible to filter the todo using a DropdownButton in the top right corner inside the AppBar.

The possible filter values are:

All (visualize completed and pending todos)

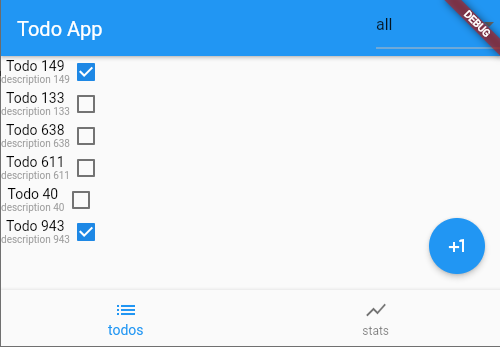
Completed (visualize completed todo)

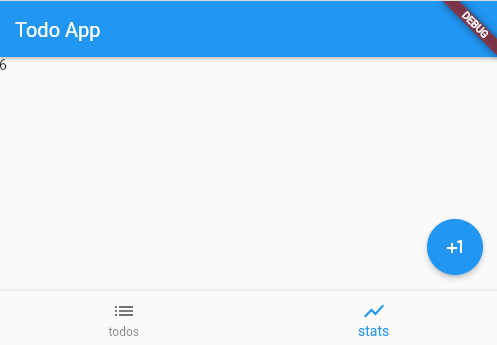
Not Completed (visualize pending todos)

The elements inside the list of todos are called TodoItems. TodoItems visualize the todo’s name and description using a Text widget and completion using a Checkbox. It is possible to use the checkbox to mark a Todo as completed or to mark it as pending.

In the stats Tab instead is possible to visualize the number of completed todos through a Text widget.

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





**Adding new features**

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

**The Add todo Feature**

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

**The Update feature**

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

**Optimize rerenders**

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

**Implementation**

**Shared project structure**

Some parts of the code will be reused between solution implementations. The complete structure of the app’s UI will be the same as the repository used to fetch the Todos

**Inherited widget and SetState implementation**

1. **Base app**

Flutter official documentation is a bit shallow about inherited widgets and how to correctly implement them. First thing first I searched a good implementation of inherited widget pattern on the web and found this article <https://stackoverflow.com/questions/49491860/flutter-how-to-correctly-use-an-inherited-widget>. It is a great discussion where [Rémi Rousselet](https://stackoverflow.com/users/8394265/r%c3%a9mi-rousselet) and  [brianegan](https://stackoverflow.com/users/8210363/brianegan) (two famous flutter programmer that have built well used packages) talk about the topic and provide a “correct” implementation.

First we need to define our InheritedData model creating a class and extendting it to InheritedWidget.

class TodoInheritedData extends InheritedWidget{

Now state’s data that needs to be accessible down the tree can be putted inside it. We create a list of Todo, a Visibilityfilter , a Int for the stats ( for cincisenes it will represent the number of completed todos) and another list of Todo that will contain the todos matching the filter. A constructor is also necessary to initialize our final variables and then compute stats and the filteredTodo list. filterTodo function is just a function I wrote that takes the full list of todos and the filter and returns the filtered list. Important to notice is the fact that a child widget must also be passed in the constructor. This is because our TodoInheritedData is nothing else than a widget itself that wraps the data and make them accessible in the child tree.

class TodoInheritedData extends InheritedWidget{

final List<Todo> todos;  
final List<Todo> filteredTodos;

final VisibilityFilter filter;

final int stats;

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

}

Is important to understand that TodoInheritedData widget is a stateless widget. It cannot be changed (every value is final) but instead a new TodoInheritedData widget must be provided when a data change occurs.

For this reason a updateShouldNotify function must be overrided inside TodoInheritedData to avoid ui rebuilding when a new state without actual data changes occurs. Once a TodoInheritedData element is replaced with a new one this new element will take care to call the updateShouldNotify function to decide whether is necessary or not to notify the subtree of the changes. If the function returns true the subtree is rebuilt, if return false instead is not.

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

the listEquals function takes as parameters the old filteredTodos list and the new one and compare them element by element to check if changes were made. In the particular case in which no changes were performed is returns true and will lead the updateShouldNotify function to return false and not to rebuild the entire subtree.

So, to make TodoInheritedData changeable we incorporate it in a stateful widget and we use setState.

class TodoProvider extends StatefulWidget {  
 const TodoProvider({Key? key, required this.child}) : super(key: key);  
  
 final Widget child;  
  
 @override  
 \_TodoProviderState createState() => \_TodoProviderState();  
}

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

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

In this way every time we call setState a new instance of TodoInheritedData is created with the value of todos and filter. ( note that the VisibilityFilter is set as “all” by default)

We add also a init method to fetch the data from the repository on the widget creation

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

loadTodos is a async function that simulate the retrieval of the todos from a database.

We need to declare also a method to retrieve our TodoInheritedData down the tree. This method is called “of”and is static and just extract the nearest TodoInheritedData element up in the tree.

static TodoInheritedData? *of*(BuildContext context) {  
 return context.dependOnInheritedWidgetOfExactType<TodoInheritedData>();  
}

At this point we can incorporate our data provider as a parent of the scaffold widget in the homepage. The usage of the Builder widget is due to the fact that data is accessible only in a context where a Todoprovider is already present. In other word we cannot use the TodoProvider’s data in the same build method where we instanciated it. We have to options; create a separated file where to put our Scaffold or use a Builder widget that takes the current context and creates another with the addional informations.

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

At this point we can populate the code inside TodoView component (a widget that takes care of showing the list of filtered todos). It is a stateless widget that will look up for the filteredTodo list of the TodoInheritedData inside the build method and create a ListView dynamically using it. The ListView will be composed by TodoItem widgets.

class TodoView extends StatelessWidget {  
  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building TodoView");  
  
 final List<Todo> filteredTodos = TodoInheritedData.*of*(context).filteredTodos;  
  
 return ListView.builder(  
 itemCount: filteredTodos.length,  
 itemBuilder: (context, index) {  
 return TodoItem(  
 todo: filteredTodos.elementAt(index),  
 );  
 },  
 );  
 }  
}

There is nothing left to do but to define the TodoItem widget and we are done. We create a stateless widget that take as paramenter a Todo and displays it. And we aare done.

class TodoItem extends StatelessWidget {  
 final Todo todo;  
  
 const TodoItem({Key? key, required this.id}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return Row(  
 children: [  
 Column(  
 children: [  
 Text(todo.name,  
 style: const TextStyle(fontSize: 14, color: Colors.*black*)),  
 Text(todo.description,  
 style: const TextStyle(fontSize: 10, color: Colors.*grey*)),  
 ],  
 ),  
 Checkbox(  
 value: todo.completed,  
 onChanged: (value) {  
 }),  
 ],  
 ),  
 );  
 }  
}

At this point we got a single page (Homepage) that contains a TodoView showing the todos present in the filteredTodos list that is contained in the TodoInheritedData inside the TodoProvider widget. when the application starts we first see and empty page (todo are empty at the beginning) and then after few seconds a list of todo with their name , description and completion appears.

For the moment we can visualize the list of fiteredTodos but we cannot do much more. Lets add the possibility to change the filter and show the completed or not completed ones instead of them all.

In the app HomePage’s AppBar we already set up a VisibilityFilterComponent that is nothing else than a stateless widget. In the build method we look up for the filter value in the TodoInheritedData and set up a DropdownButton value field with it. Then in the items field we pass a list of DropdownMenuItem that comes from the mapping of all possible VisibilityFilter values to DropdownMenuItems.

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

For what concerns the onChanged field we need to provide a function that takes as single parameter a filter value and do something. In particular we want this function to change the state contained in the TodoInheritedData (the filter part) and to fire a rebuild of the corresponding subtree. As we mentioned above TodoInheritedData contains only final field and should never be modified. Instead we want to create a new TodoInheritedData element and set the state of the TodoProvider statefull widget with it.

To do so we go back to TodoProvider.dart and add a function called onChangeFilter that takes the new filter values as parameter in the TodoProvider statefull widget.

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

this function changes the value of the filter in the statefull widget calling setState and doing so the build function is called again with the new filter data and a new TodoInheritedData widget is created.

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

As can be noticed the actual function must be provided to the TodoInheritedData to make it accessible in the widgets in the subtree. For this reason we add new parameter in the TodoInheritedData as follow.

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

Now that we have the changing filter function accessible down in the tree we can simply call it in the function we provide inside the VisibilityFilterComponent DropdownButton onChange field.

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

We can now display the filteredTodo list and also change it appling different filters. However the Checkbox inside every TodoItem for the moment is just show if the particular todo is completed or not but its onChange function is empty and does nothing. We want the Checkbox to change the Todo completed field in the app state when clicked and to rebuild itself to show the actual change ( for the moment we don’t care if the TodoItem only is rebuild or the entire TodoView). To do so we need to provide a function down the tree that allow to perform this change. Going back again to the TodoProvider.dart file a onSetCompleted function is added to the statefull widget. This function take as parameter the id of the Todo to be changed and the new value for the completed field.

void onSetCompleted(int id, bool completed) {  
 assert(todoExists(id) != null, 'No todo with id : $id');  
  
 setState(() {  
 todos = todos.map((e) {  
 if (e.id == id) {  
 return Todo(  
 id: id,  
 name: e.name,  
 description: e.description,  
 completed: completed);  
 } else {  
 return e;  
 }  
 }).toList();  
 });  
}

Using a map the todos list is scanned. Once the todo with the corresponding id is found its completed value is changed to the newValue. Like before, calling the setState method on the statefull widget will cause to create another TodoInheritedData element and to rebuild the entire subtree under TodoProvider widget.

At this point all the vbasic functionalities are working fine. It required about 2-3 hours of programming to get the job done without errors nor bugs.

1 class and 1 stateful widget

* 1. Files

13 file in total. 1 for the state, and 12 for the base

1. Add features
   1. Line of code

18

Immagine che contiene testo

Descrizione generata automaticamente

TodoInheritedData.*of*(context).onAddTodo();

* 1. Time and effort

Low effort and low time. Just add a method and call it from homepage

* 1. Components refactoring

None

1. Update feature

61

Low and low ,about 30 miuntes.

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

InkWell(  
 onTap: () {  
 Navigator.*pushNamed*(context, "/updateTodo",  
 arguments: UpdateTodoPageArguments(  
 todo: todo,  
 updateState: (String newName) {  
 TodoInheritedData.*of*(context, aspect: 0)  
 .onSetName(todo.id, newName);  
 }));  
 },

class UpdateTodoPageArguments {  
 final Todo todo;  
 final void Function(String newName) updateState;  
  
 UpdateTodoPageArguments({required this.todo, required this.updateState});  
}

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

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

TextButton(onPressed: () {  
  
 widget.callback(textController.text);  
 Navigator.*pop*(context);  
},

1. Widget rebuild optimization
   1. Time and effort

About 1 day of work and a lot of effort  
Was really hard to understand how to do it and to find the correct material/help on the internet

I needed to migrate to inheritedmodel

Write a specific method to register every TodoItem to the changes of the respective todo in the list

From this

//static TodoInheritedData? of(BuildContext context) {  
// return context.dependOnInheritedWidgetOfExactType<TodoInheritedData>();  
//}

To

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

* 1. Line of code
  2. Component refactoring

Yes, I need to code again the todoview in order to make todoitems connected with the state directly and not be created by the todoview local state. Indeed if we write something like :

class TodoItem extends StatelessWidget {  
 final Todo todo;  
  
 const TodoItem({Key? key, required this.todo }) : super(key: key);

If the data in the TodoInherited state change we need to rebuild the entire todoview to make single todoitem change. This because if we pass the todo as a constructor parameter we are creating a copy of it and pass it to che todoitem. So if we change the todo in the central state the todoItem’s local todo will not upadate/ change and the todoItem will rebuild with the same information creating a bad behaviour where the displayed data do not refeclect the real one. So instead of passing a copy of the todo we should pass only the id and look up for the todo in the todoItem in this way:

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

so now is the widget is rebuilt the data displayed will be reflecting the real one. At this point we have the desired behaviour and we can tri to achive the partial rebuilding we want. With simple inherited widget IS NOT POSSIBLE to obtain this behaviour. This because every time the data changes ( also a really small part) all the InheritedWidget subtree is rebuilt. We can only choose if we want to rebuild it or not with the

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

If this function return true the entire subtree will be rebuilt, instead is it returns false it is not. In this particular case the function listEquals compare the old state filtered todo list with the new one and return true only when they are different. listEquals check recursively the equality of the lists. Will return true only if the two list contains the exact same elements checking also the == operator todo by todo.

This is not enough tough. With this method we can also decide to rebuild the entire tree or not. To have a partial rebuild we need to migrate to InheritedModel that was made to handle exactly this behaviour.

First we need to change out Inheriteddata to extends inheritedModel instead of inheritedWidget

from

class TodoInheritedData extends InheritedWidget {

to

class TodoInheritedData extends InheritedModel<int> {

then change our OF function from the simple one:

//static TodoInheritedData? of(BuildContext context) {  
// return context.dependOnInheritedWidgetOfExactType<TodoInheritedData>();  
//}

To the inheritedMmodel one where a new parameter is required.

This is because once a widget call the Of function it automatically register to the state changes and will rebuild on them. With this new of function we pass also a “aspect” parameter ( in this case a int for conciseness) that inform the of method on which type of changes the calling widget is interested into and register to them. A widget could be registered for changes of multiple aspects but in this Todo app example every widget will register to only one aspect of the data.

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

A this point we have multiple widget looking at the state and we need a function to decide which one of them notify on a state change. We have to override the updateshouldnotifydependent inheritedmodel function:

@override  
bool updateShouldNotifyDependent(  
 TodoInheritedData oldWidget, Set<int> dependencies)

{. . . }

This was a short function to write but cost me a lot of effort and preciseness to code it correctly.

It is called for every widget that called the of method with the same oldWidget parameter and in the dependencies parameter the set of dependencies it registered for (in our case only one).  
At this point when we change the state we will have the execution of this function once for the todoView widget and once for every todoItem widget.

In the todoView we access the inherited data like this :

final List<Todo> filteredTodos = TodoInheritedData.*of*(context, aspect: 0).filteredTodos;

registering for the aspect 0 (I decided to map the 0 with the structure change of the todo list)

in the todoItem instead we access the state like this :

final Todo todo = TodoInheritedData.*of*(context, aspect: id)  
 .todos  
 .where((element) => element.id == id)  
 .first;

registering for changes in the “id” aspect meaning that we register only to changes in the Todo with the TodoItem id.

This is the updateshoudlsdgd… implementation:

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

It is a complicated function so I will try to explain it with two examples that exauste the possible scenarios.

Example 1:

We are executing the function for the TodoView widget.

So in this case the set of dependencies contains the int 0.( this because we previously registered to it).

Now the function can return true only if the filteredTodo list has changed structure. With structure changes I mean that the length of the neew filteredTodo list in diffent from the previous one or the todo’s id inside it are changed. So if the data change was originated by a new todo insertion fro example the todoView will rebuild but instead will not rebuild if the data change was inside a particular widget, for example we changed the filed “completed” of the widget with id 3. In this case both the length and the id contained did not change so we don’t need to rebuild the entire TodoView.  
some possible scenarios:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | oldWidget.filteredTodos | filteredTodos | Return value |
| Field change in a particular todo | [1,2,4] | [1,2,4] | False |
| New todo insertion | [1,2] | [1,2,3] | True |
| Todo replacement with a new one | [1,3,2] | [1,4,2] | True |

Example 2:

We are execution the function for the TodoItem with id 3.

So in this case the set of dependencies contains the int 3.( this because we previously registered to it in the TodoItem). Now suppose the data change was fired by the TodoItem with id 4. This means that the function knows we changed the aspect identified with the number 4. In this case the dependencies contain only the aspect 3 so the TodoItem Widget with id 4 is not rebuilt. In the execution of the dsndkhfanaskn function for the TodoItem widget with id 4 instead there is a match with the aspect changed and the TodoItem is rebuild.

Conclusion : about 25-30 lines of code , a pretty hard function to code and a day of work