Todo app development

Inherited widget and SetState

1. Base app
   1. Development time

Fast , couple of hours for a correctly working demo

* 1. Line of code

(460)?

* 1. Additional classes/widgets needed

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