# 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 measurements. In particular for every solution three development processes will be executed. First will be implemented the basic functionalities of the app. Then I will measure how much effort/code is needed to add other functionalities. In the last round some optimization will be made to the code in terms of UI renders and memory consumption.

# The Todo app

The first app presented is a simple Todo app.

## General overview

The development process is divided in three parts.

### 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.

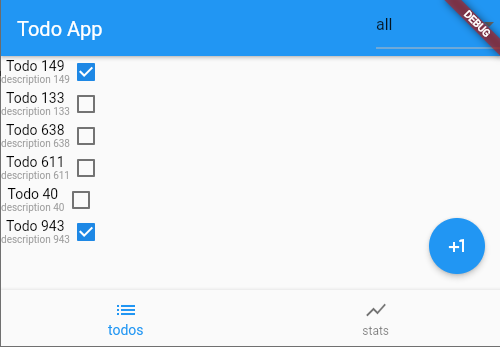
The possible filter values are:

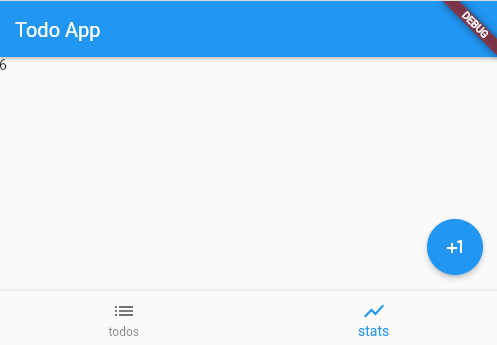
* **All** (visualize completed and pending todos)
* **Completed** (visualize completed todo)
* **Not Completed** (visualize pending todos)

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.

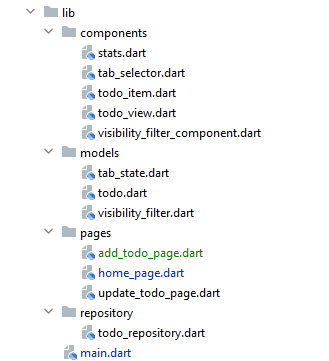
### Renders optimization

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

## Implementation

### Shared project structure and files

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



The figure 2.2.3 illustrate the folder and files structure.

#### main.dart

void main() {  
 runApp(const MyApp());  
}

class MyApp extends StatefulWidget {  
 const MyApp({Key? key}) : super(key: key);  
  
 @override  
 State<MyApp> createState() => \_MyAppState();  
}  
  
class \_MyAppState extends State<MyApp> {  
  
 @override  
 Widget build(BuildContext context) {  
 return MaterialApp(  
 initialRoute: "/",  
 routes: {  
 "/": (context) => const HomePage(),  
 "/updateTodo": (context) => UpdateTodoPage(),

"/addTodo": (context) => AddTodoPage(),  
  
 },  
 );  
 }  
}

The entire app is wrapped into a MaterialApp widget with three routes. The HomePage , the UpdateTodoPage and the AddTodoPage.

#### Models and Repository

**tab\_state.dart**

enum TabState{  
 todos,stats  
}

Tab states represent the possible tabs in the homepage.

**visibility\_filter.dart**

enum VisibilityFilter{

completed,notCompleted,all

}

**todo.dart**

@immutable  
class Todo {  
 final int id;  
 final String name;  
 final String description;  
 final bool completed;  
  
 const Todo(  
 {required this.id,  
 required this.name,  
 required this.description,  
 required this.completed});  
  
 @override  
 bool operator ==(Object other) {  
 return (other is Todo) &&  
 other.description == description &&  
 other.name == name &&  
 other.id == id &&  
 other.completed == completed;  
 }  
  
 @override  
 String toString() {  
 return "{ id: $id completed: $completed}";  
 }  
  
 @override  
 *// TODO: implement hashCode* int get hashCode => super.hashCode;  
}

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

**todo\_repository.dart**

class TodoRepository {  
 static Future<List<Todo>> *loadTodos*() async {  
 Random rand = Random();  
 List<Todo> todos = [];  
 List<int> ids = [];  
 while (ids.length < 6) {  
 int newInt = rand.nextInt(1000)+2;  
 if (!ids.contains(newInt)) {  
 ids.add(newInt);  
 }  
 }  
 todos = ids  
 .map((number) => Todo(  
 id: number,  
 name: "Todo " + number.toString(),  
 description: "description " + number.toString(),  
 completed: rand.nextBool()))  
 .toList();  
  
 await Future.delayed(const Duration(seconds: 2));  
 return todos;  
 }  
  
 static Future<void> *saveTodos*(List<Todo> todos) async {  
 await Future.delayed(const Duration(seconds: 2));  
 }  
}

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

#### Pages

**home\_page.dart**

class HomePage extends StatefulWidget {  
 const HomePage({Key? key}) : super(key: key);  
  
 @override  
 State<HomePage> createState() => \_HomePageState();  
}  
  
class \_HomePageState extends State<HomePage> {  
 TabState tab = TabState.todos;   
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 actions: [  
 tab == TabState.todos  
 ? const VisibilityFilterComponent()  
 : Container()  
 ],  
 title: const Text("Todo App"),  
 ),  
 body: tab == TabState.todos ? const TodoView() : const Stats(),  
 bottomNavigationBar: TabSelector(  
 currTab: tab,  
 onTabChange:,  
 ),  
 floatingActionButton: tab == TabState.todos  
 ? FloatingActionButton(  
 child: const Icon(Icons.*plus\_one*),  
 onPressed: () {},  
 ) : null,  
 )  
 );  
 }  
}

Homepage uses a simple Scaffold widget. The AppBar contains a VisibilityFilterComponent only when the tab is set to Todos. The body can change from todos to stats tab using the bottomNaviagationBar TabSelector. An empty FloatingActionButton is present for future implementation.

(note: some small pieces could change in different solution’s implementation. in the above example the tab changing is implemented through setState but it will not be always the case. Also the HomePage can be muted to Stateless widget in other implementations.).

**update\_todo\_page.dart**

class UpdateTodoPage extends StatefulWidget {  
 final Todo todo;  
  
 const UpdateTodoPage({Key? key, required this.todo}) : super(key: key);  
  
 @override  
 State<UpdateTodoPage> createState() => \_UpdateTodoPageState();  
}  
  
class \_UpdateTodoPageState extends State<UpdateTodoPage> {  
 final textController = TextEditingController();  
  
 @override  
 Widget build(BuildContext context) {  
  
 return Scaffold(  
 appBar: AppBar(  
 title: const Text("Update Todo"),  
 ),  
 body: Column(  
 children: [  
 TextField(  
 controller: textController,  
 decoration: const InputDecoration(  
 border: OutlineInputBorder(), hintText: 'Enter a new name'),  
 ),  
 TextButton(onPressed: (){}, child: const Text("Confirm"))  
 ],  
 ));  
 }  
  
 @override  
 void dispose() {  
 textController.dispose();  
 super.dispose();  
 }  
}

The update todo page uses a Scaffold widget. The body is composed by a Column with inside a TextField and a TextButton. The TextButton is left empty for future implementation.

**add\_todo\_page.dart**

class AddTodoPage extends StatefulWidget {  
  
 const AddTodoPage({Key? key, required this.addTodoCallback}) : super(key: key);  
  
 @override  
 State<AddTodoPage> createState() => \_UpdateTodoPageState();  
}  
  
class \_UpdateTodoPageState extends State<AddTodoPage> {  
 final textControllerName = TextEditingController();  
 final textControllerDesc = TextEditingController();  
  
 @override  
 Widget build(BuildContext context) {  
  
 return Scaffold(  
 appBar: AppBar(  
 title: const Text("Add Todo"),  
 ),  
 body: Column(  
 children: [  
 TextField(  
 controller: textControllerName,  
 decoration: const InputDecoration(  
 border: OutlineInputBorder(), hintText: 'Enter a name'),  
 ),  
 TextField(  
 controller: textControllerDesc,  
 decoration: const InputDecoration(  
 border: OutlineInputBorder(), hintText: 'Enter a description'),  
 ),  
 TextButton(onPressed: (){},

child: const Text("Create"))  
 ],  
 ));  
 }  
  
 @override  
 void dispose() {  
 textControllerName.dispose();  
 textControllerDesc.dispose();  
 super.dispose();  
 }  
}

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

#### Components

**todo\_view.dart**

class TodoView extends StatelessWidget {  
  
 const TodoView({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building TodoView");  
  
  
 return ListView.builder(  
 itemCount:,  
 itemBuilder: (context, index) {  
 return TodoItem(  
   
 );  
 },  
 );  
 }  
}

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

**todo\_item.dart**

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

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

**tab\_selector.dart**

class TabSelector extends StatelessWidget {  
  
  
 const TabSelector(  
 {Key? Key})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Tab Selector");  
  
 return BottomNavigationBar(  
 currentIndex: ,  
 onTap: (){},  
 items: TabState.values  
 .map((tab) => BottomNavigationBarItem(  
 label: describeEnum(tab),  
 icon: Icon(  
 tab == TabState.todos ? Icons.*list* : Icons.*show\_chart*,  
 ),  
 ))  
 .toList(),  
 );  
 }  
}

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

**visibility\_filter\_component.dart**

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

VisibilityFilterComponent uses a DropdownButton with as many DropwodnMenuItems as VisibilityFilter.values (in our case two). Function fields are left empty for future implementation.

**stats.dart**

class Stats extends StatelessWidget {  
 const Stats({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Stats");  
  
 return Text();  
 }  
}

Stats component is only a Text widget showing stats value.

### Inherited widget/model and SetState implementation

In this section Todo app will be implemented using two standard feature Flutter framework provides to ha handle state: Inherited widget (or the more advanced InheritedModel) and setState.

#### State management solution’s introduction

**setState** method notify the framework that the internal state of this object has changed.

Whenever you change the internal state of a [State](https://api.flutter.dev/flutter/widgets/State-class.html) object, make the change in a function that you pass to [setState](https://api.flutter.dev/flutter/widgets/State/setState.html).

setState(() { \_myState = newValue; });

The provided callback is immediately called synchronously. It must not return a future (the callback cannot be async), since then it would be unclear when the state was actually being set.

Calling [setState](https://api.flutter.dev/flutter/widgets/State/setState.html) notifies the framework that the internal state of this object has changed in a way that might impact the user interface in this subtree, which causes the framework to schedule a [build](https://api.flutter.dev/flutter/widgets/State/build.html) for this [State](https://api.flutter.dev/flutter/widgets/State-class.html) object.

If you just change the state directly without calling [setState](https://api.flutter.dev/flutter/widgets/State/setState.html), the framework might not schedule a [build](https://api.flutter.dev/flutter/widgets/State/build.html) and the user interface for this subtree might not be updated to reflect the new state.

**Inherited widget** are a base class for widgets that efficiently propagate information down the tree.

To obtain the nearest instance of a particular type of inherited widget from a build context, use *[BuildContext.dependOnInheritedWidgetOfExactType](https://api.flutter.dev/flutter/widgets/BuildContext/dependOnInheritedWidgetOfExactType.html).*

Inherited widgets, when referenced in this way, will cause the consumer to rebuild when the inherited widget itself changes state.

The convention is to provide a static method *of* on the [InheritedWidget](https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html) which does the call to [BuildContext.dependOnInheritedWidgetOfExactType](https://api.flutter.dev/flutter/widgets/BuildContext/dependOnInheritedWidgetOfExactType.html). This allows the class to define its own fallback logic in case there isn't a widget in scope. In the example above, the value returned will be null in that case, but it could also have defaulted to a value.

An [InheritedWidget](https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html) that's intended to be used as the base class for models whose dependents may only depend on one part or "aspect" of the overall model. An inherited widget's dependents are unconditionally rebuilt when the inherited widget changes.

**InheritedModel** widget is similar except that dependents aren't rebuilt unconditionally.

Widgets that depend on an [InheritedModel](https://api.flutter.dev/flutter/widgets/InheritedModel-class.html) qualify their dependence with a value that indicates what "*aspect*" of the model they depend on. When the model is rebuilt, dependents will also be rebuilt, but only if there was a change in the model that corresponds to the aspect they provided.

#### Base app

There is not a cleas implementation guide on the official Flutter documentation. Some development structure and techniques of the follow part are taken from <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" \o "386 reputation) (two famous flutter programmer that have built well used packages) talk about the topic and provide a “correct” implementation.

First thing first our InheritedData model should be defined creating a class and extendting it to InheritedWidget.

class TodoInheritedData extends InheritedWidget{

Data that should be accessible down the tree must be placed inside it. In our case those data are: a list of Todo, a Visibilityfilter , a Int for the stats ( for conciseness it will represent the number of completed todos) and another list of Todo that will contain the todos matching the filter. Inside the constructor final variables are initialized and stats and filteredTodo list are computed. *filterTodo* function is just a function that takes the full list of todos and a filter and returns the filtered list. Important to notice is the fact that a *child* widget must 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 overridden inside TodoInheritedData to avoid ui rebuilding when a new state without actual data changes occurs. Once a TodoInheritedData element is replaced with a new one this new element will take care to call the updateShouldNotify function to decide whether is necessary or not to notify the changes in the subtree. If the function returns *true* the subtree is rebuilt, if return *false* instead is not.

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

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

At this point our TodoInheritedData can be used it in a stateful widget and once the setState method is called a new instance will be created.

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

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

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

Note that the VisibilityFilter is set as “all” by default as a convention.

We add also an init method to fetch the data from the repository on widget’s creation.

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

*loadTodos* is a TodoRepository’s async function that simulate the retrieval of the todos from a database.

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

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

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

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

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

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

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

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

At this point we got a single page (Homepage) that contains a TodoView showing filteredTodos list’s todos contained in the TodoInheritedData inside a TodoProvider widget. When the application starts we first see and empty page (todo are empty at the beginning) and then after few seconds a list of todo with their name , description and completion appears. A list of fitered todos can be visualized but is not interactable yet.

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

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

For what concerns the onChanged field a function that takes as single parameter a filter value must be provided. In particular we want this function to change the state contained in the TodoInheritedData (the filter part) and to fire a rebuild of the TodoInheritedData subtree. As we mentioned above TodoInheritedData contains only final field and should never be modified. Instead, a new TodoInheritedData element should be created in the TodoProvider build method with the modified data.

In the to TodoProvider.dart a function called onChangeFilter is added. This function takes the new filter values as parameter and changes the value of the filter in the statefull widget calling setState. Doing so the build function is called again with the new filter value and a new TodoInheritedData widget is created.

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

The actual function must be provided to the TodoInheritedData to make it accessible in the widget’s in the subtree. To do so a new parameter is added in the TodoInheritedData as follow.

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

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

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

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

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

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

The todos list is scanned using a map. Once the todo with the corresponding id is found its completed value is changed to the newValue. Calling the setState method on the TodoProvider stateful widget will cause the build method to run again and to create another TodoInheritedData element rebuilding the entire subtree under TodoProvider widget.

At this point is possible to visualize the filteredTodo list, change the filter and update Todo’s completed value. To implement the stats tab the stats component must be connected to data and TabSelector logic defined. First, the stats value is retrieved in the stats component widget using the of method and injected in the Ui.

class Stats extends StatelessWidget {  
 const Stats({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Stats");  
  
 return Text(TodoInheritedData.*of*(context).stats.toString());  
 }  
}

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

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

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

class TabSelector extends StatelessWidget {  
 final TabState currTab;  
 final Function(int) onTabChange;  
  
 const TabSelector(  
 {Key? key, required this.currTab, required this.onTabChange})  
 : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 print("Building Tab Selector");  
  
 return BottomNavigationBar(  
 currentIndex: TabState.values.indexOf(currTab),  
 onTap: onTabChange,  
 items: TabState.values  
 .map((tab) => BottomNavigationBarItem(  
 label: describeEnum(tab),  
 icon: Icon(  
 tab == TabState.todos ? Icons.*list* : Icons.*show\_chart*,  
 ),  
 ))  
 .toList(),  
 );  
 }  
}

At this point all the basic functionalities have been implemented.

Time spent: 2-3 hours

Lines of code written/updated: 86

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

#### Features addition

**Todo addition feature :**

First thing is to make the addTodo feature/function accessible down the tree. A new function must be created in the TodoProvider widget and passed to the TodoInheritedData widget.

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

this new function will be called onAddTodo and will take two parameters (name and description). After generating a new unique id it creates a new Todo with the completed field set to false. Adding the new Todo to the stateful’s todos list requires a bit of workaround. The state of a stateful widget is immutable. It can only be changed by the setState method. Unfortunately the method “add” for lists is of type void and do not return a new list but instead add the new value to the existing one. For this reason calling directly todos.add(newTodo); inside the setState method will have no effect.

Stateful’s todos list must be completely replaced with a new list containing the also the new todo. First a newList is created and populated with the element present in the todos one. Then the newTodo is added to the newList list. At this point is sufficient to replace the todos list with the new one inside the setState method.

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

final void Function(String,String) onAddTodo;

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);  
},

#### Render optimizations

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