

**SEB-312 Mobile Application Development**

**LAB # 8**

**LAB Title**

|  |
| --- |
| Implementing local state management in a to-do list app. Using Provider to manage global state in counter app. |

**Assessment of CLO: 03, PLO: 05**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name:** |  | | |
| **Roll No.** |  | | |
| **Semester** |  | **Session** |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S. No.** | **Perf. Level**  **Criteria** | **Excellent**  **(2.5)** | **Good**  **(2)** | **Satisfactory**  **(1.5)** | **Needs Improvement**  **(0 ~ 1)** | **Marks Obtained** |
| **1** | Project Execution & Implementation | Fully functional, optimized, and well-structured. | Minor errors, mostly functional. | Some errors, requires guidance. | Major errors, non-functional, or not Performed. |  |
| **2** | Results & Debugging  Or Troubleshooting | Accurate results with effective debugging  Or Troubleshooting. | Mostly correct, some debugging Or Troubleshooting needed. | Partial results, minimal debugging  Or Troubleshooting. | Incorrect results, no debugging Or Troubleshooting, or not attempted. |  |
| **3** | Problem-Solving & Adaptability  (VIVA) | Creative approach, efficiently solves challenges. | Adapts well, minor struggles. | Some adaptability, needs guidance. | Lacks innovation or no innovation, unable to solve problems. |  |
| **4** | Report Quality & Documentation | Clear, structured, with detailed visuals. | Mostly clear, minor gaps. | Some clarity issues, missing details. | Poorly structured, lacks clarity, or not submitted. |  |
| **Total Marks Obtained Out of 10** | | | | | |  |

**Experiment evaluated by**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructor’s Name** | **Sidra Khatoon** | | |
| **Date** |  | **Signature** |  |

**Objective**

The objective of lab is implementing local state management in a to-do list app. Also using Provider to manage global state in counter app

**Instructions**

You have to perform the following tasks yourselves. Raise your hand if you face any difficulty in understanding and solving these tasks. **Plagiarism** is an abhorrent practice and you should not engage in it.

**Learn Local State Management in Flutter by Building a Simple Todo App:**

**Learn State Management in Flutter by Building a Simple Todo App:**

State management is a complex topic in mobile application development. But it's also a necessary topic that plays a major role in building dynamic mobile apps.

First we create new app with basic counter application. Rename class name of MyHomePage with TodoList. Also remove comments and extra code from app.

Let's build the core functionality of our app.

We need a Todo class. This class will define the properties of a todo. In our case, we'll have the following items:

1. Name of the todo
2. Status of the todo (Completed)

Let's define a Todo class with the above properties:

class Todo {

    String name;

    bool completed;

Todo({required this.name, required this.completed});

}

Add the above code at the bottom of the main.dart file.

**How to Add a Todo**

In the body of the build method, set the children property to an empty array. Remove the two Text widgets inside that children property. Now we'll replace the counter variable with a todo list. Repace the code int \_counter=0; with the below code:

final List<Todo> \_todos = <Todo>[];

final TextEditingController \_textFieldController = TextEditingController();

The first line is the todo list and the second line defines the controller to get the name of the todo from the user.

Remove the \_incrementCounter method and add the method to add a todo:

  void \_addTodoItem(String name) {

      setState(() {

          \_todos.add(Todo(name: name, completed: false));

      });

      \_textFieldController.clear();

  }

So far we have defined our todo list and an input controller. We've also created a method that accepts input text and adds that to the todo list with a completed status set to false and a clear input field.

The reason we have used the setState method is to refresh the UI after we update the todo list. As our component is a stateful widget, whenever a change in state is detected, the UI will render again with the updated state.

We have built the functionality code to add a todo. Let's build the UI code. Let's ask the user the name of the todo on pressing the Floating action button at the bottom right. When the user tries to save the todo, we'll call the \_addTodoItem method defined above. We replace counter floating button code with this code:

 floatingActionButton: FloatingActionButton(

        onPressed: () => \_displayDialog(),

        tooltip: 'Add a Todo',

        child: const Icon(Icons.add),

      ),

In the above method, we have changed the onPressed property to call the \_displayDialog method. As it's not defined yet, it'll show an error. We'll define the method next. We have also changed the tooltip property to "Add a Todo".

Here's the code (\_displayDialog method) to show a dialog box with an input field, add, and cancel button. Add this method inside the \_MyHomePageState class:

Future<void> \_displayDialog() async {

    return showDialog<void>(

        context: context,

        barrierDismissible: false,

        builder: (BuildContext context) {

            return AlertDialog(

                title: const Text('Add a todo'),

                content: TextField(

                    controller: \_textFieldController,

                    decoration: const InputDecoration(hintText: 'Type your todo'),

                    autofocus: true,

                ),

                actions: <Widget>[

                    OutlinedButton(

                        style: OutlinedButton.styleFrom(

                            shape: RoundedRectangleBorder(

                                borderRadius: BorderRadius.circular(12),

                            ),

                        ),

                        onPressed: () {

                            Navigator.of(context).pop();

                        },

                        child: const Text('Cancel'),

                    ),

                    ElevatedButton(

                        style: ElevatedButton.styleFrom(

                        shape: RoundedRectangleBorder(

                       borderRadius: BorderRadius.circular(12),

                     ),

                 ),

                  onPressed: () {

                        Navigator.of(context).pop();

                        \_addTodoItem(\_textFieldController.text);

                  },

                  child: const Text('Add'),

                  ),

             ],

            );

        },

      );

    }

In our case, it'll wait for the user to tap the Add or Cancel button. The \_displayDialog method will return the showDialog method by building the UI.

The barrierDismissible property is used to define if the pop up has to be closed if the user taps outside of the alert dialog. We have set that to false which means the alert dialog will not be closed on taping outside.

The builder of this showDialog method returns an AlertDialog consisting of title, content, and actions property. The title is set to display the text "Add a todo". The content property will render an text input field with automatic focus enabled and the hint "Type your todo".

The actions property will render 2 buttons, Cancel and Add. The Cancel button is an outlined button, and pressing it will close the dialog. The Add button adds the text to the todo list and closes the dialog.

**How to List the Todos**

We have added the code to add todos to the list. But wait – how can we verify that? We have to find if the todo has actually been added to the list.

Let's verify that by rendering the list of todo items in the UI. To do so, we have to design the UI for a single todo. Let's do that, add the following code at the end of main.dart file:

class TodoItem extends StatelessWidget {

    TodoItem({required this.todo}) : super(key: ObjectKey(todo));

    final Todo todo;

    TextStyle? \_getTextStyle(bool checked) {

        if (!checked) return null;

        return const TextStyle(

            color: Colors.black54,

            decoration: TextDecoration.lineThrough,

        );

    }

    @override

    Widget build(BuildContext context) {

        return ListTile(

            onTap: () {},

            leading: Checkbox(

                checkColor: Colors.greenAccent,

                activeColor: Colors.red,

                value: todo.completed,

                onChanged: (value) {},

            ),

            title: Row(

                children: <Widget>[

                    Expanded(

                        child: Text(

                            todo.name,

                            style: \_getTextStyle(todo.completed),

                        ),

                    ),

                    IconButton(

                        iconSize: 30,

                        icon: const Icon(

                            Icons.delete,

                            color: Colors.red,

                        ),

                        alignment: Alignment.centerRight,

                        onPressed: () {},

                    ),

                ],

            ),

        );

    }

}

First, we created a class with the TodoItem and we extended it from the StatelessWidget class as we don't need to maintain state for this class. We accept a Todo, which is passed via constructor to our class. The code in the build method determines the UI. It renders the ListTile widget with the Checkbox widget passed to the leading property.

The title property renders a row of Text and IconButton widgets. The Text widget shows the name of the todo and the IconButton widget displays the delete icon. **Notice the \_getTextStyle method passed to the style property of the Text widget. This method strikes out the text if the todo is marked as complete.** Nothing changes on tapping any of these widgets, as the corresponding properties are left empty (onTap, onChanged, and onPressed).

Change the body property of the build method in \_TodoListState with the following code:

  body: ListView(

          padding: const EdgeInsets.symmetric(vertical: 8.0),

          children: \_todos.map((Todo todo) {

              return TodoItem(

                  todo: todo,

              );

          }).toList(),

      ),

The above code defines a ListView widget iterating over the created todos list and passing each todo list to the TodoItem widget. We're done with listing the todos. Let's verify if both creating and viewing a todo works fine.

**How to Update a Todo**

Let's mark the todo as complete on pressing the checkbox near each todo list. We have 2 fields in our Todo class. They're name and completed status. Whenever a Todo is created, the default value of the completed field is set to false. This means the todo is in progress. We can change that to true whenever we complete the task.

Define a method called \_handleTodoChange in the \_TodoListState class. Add this method below the \_addTodoItem method which we defined to add a todo to the list.

void \_handleTodoChange(Todo todo) {

      setState(() {

          todo.completed = !todo.completed;

      });

  }

In the above code, we accept a todo and change the completed status of the todo. So, whenever this method is called with a todo, it's completed status will change from true to false or vice versa. Remember that we have wrapped this inside a setState method to render the UI after making the change.

We have to trigger this method when a user taps on a todo or taps on a checkbox. We should pass this method to the TodoItem class. While calling the TodoItem in the build method of the \_TodoListState class, pass the \_handleTodoChange method as shown below:

  return TodoItem(

                  todo: todo,

                  onTodoChanged: \_handleTodoChange

              );

As we're passing the method to the TodoItem class, we should receive the same method in the TodoItem class. To do so, we have to define this method in the constructor of the TodoItem class. Go to TodoItem and change the constructor to include the onTodoChanged method.

 TodoItem({required this.todo, required this.onTodoChanged,}) : super(key: ObjectKey(todo));

You may notice in the above code that we use \*\*this\*\*.onTodoChanged, which means we're binding the method passed to a method in this TodoItem class.

Let's define a method with the same name and set the return type to void (as we don't expect anything from that method).

final void Function(Todo todo) onTodoChanged;

So, wherever we call this method in our code, the status of our todo will be changed to the opposite. Let's call this method in the onTap property of the ListTile widget and onChanged property of the Checkbox widget.

return ListTile(

            onTap: () {

              onTodoChanged(todo);

            },

            leading: Checkbox(

                checkColor: Colors.greenAccent,

                activeColor: Colors.red,

                value: todo.completed,

                onChanged: (\_) => onTodoChanged(todo),

            ),

**How to Delete a Todo**

We have only one item left to complete this app. We should be able to delete a todo, if we create one by mistake or if it's no longer applicable.

Steps to delete a todo are similar to updating a todo. We'll doing the exact 4 steps as we did for updating a todo.

1. Define the \_deleteTodo method

2. Pass the method on TodoItem render

3. Receive the method on TodoItem constructor

4. Bind the method

5. Call the method on button tap

Here's the method to delete the todo. Add this in the \_TodoListState class below the \_handleTodoChange method:

 void \_deleteTodo(Todo todo) {

     setState(() {

         \_todos.removeWhere((element) => element.name == todo.name);

     });

  }

This method accepts a todo, compares it with the todo list, and identifies the todo which matches with this name. Then it deletes it from the list and finally updates the state.

final void Function(Todo todo) removeTodo;

Let's pass the method reference to TodoItem in the build method of the \_TodoListState class.

return TodoItem(

                  todo: todo,

                  onTodoChanged: \_handleTodoChange,

                  removeTodo:\_deleteTodo

                );

Also change constructor code in TodoItem to accept removeTodo

TodoItem({required this.todo, required this.onTodoChanged,}) : super(key: ObjectKey(todo));

in TodoItem class with this code:

 TodoItem({required this.todo, required this.onTodoChanged,required this.removeTodo}) : super(key: ObjectKey(todo));

Also add this variable var removeTodo; in TodoItem Class.

Our final step is to call this method on pressing the delete button.

 IconButton(

            iconSize: 30,

            icon: const Icon(Icons.delete, color: Colors.red),

            onPressed: () => removeTodo(todo),

          ),

**Our Final Code of Todo Application:**

import 'package:flutter/material.dart';

void main() {

  runApp(const MyApp());

}

class MyApp extends StatelessWidget {

  const MyApp({super.key});

  // This widget is the root of your application.

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      debugShowCheckedModeBanner: false,

      title: 'Todo App',

      theme: ThemeData(

        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),

        useMaterial3: true,

      ),

      home: const TodoList(),

    );

  }

}

class TodoList extends StatefulWidget {

  const TodoList({super.key});

  @override

  State<TodoList> createState() => \_TodoListState();

}

class \_TodoListState extends State<TodoList> {

  final List<Todo> \_todos = <Todo>[];

  final TextEditingController \_textFieldController = TextEditingController();

  void \_addTodoItem(String name) {

      setState(() {

          \_todos.add(Todo(name: name, completed: false));

      });

      \_textFieldController.clear();

  }

  void \_handleTodoChange(Todo todo) {

      setState(() {

          todo.completed = !todo.completed;

      });

  }

  void \_deleteTodo(Todo todo) {

     setState(() {

         \_todos.removeWhere((element) => element.name == todo.name);

     });

  }

  Future<void> \_displayDialog() async {

    return showDialog<void>(

        context: context,

        barrierDismissible: false,

        builder: (BuildContext context) {

            return AlertDialog(

                title: const Text('Add a todo'),

                content: TextField(

                    controller: \_textFieldController,

                    decoration: const InputDecoration(hintText: 'Type your todo'),

                    autofocus: true,

                ),

                actions: <Widget>[

                    OutlinedButton(

                        style: OutlinedButton.styleFrom(

                            shape: RoundedRectangleBorder(

                                borderRadius: BorderRadius.circular(12),

                            ),

                        ),

                        onPressed: () {

                            Navigator.of(context).pop();

                        },

                        child: const Text('Cancel'),

                    ),

                    ElevatedButton(

                        style: ElevatedButton.styleFrom(

                        shape: RoundedRectangleBorder(

                       borderRadius: BorderRadius.circular(12),

                     ),

                 ),

                  onPressed: () {

                        Navigator.of(context).pop();

                        \_addTodoItem(\_textFieldController.text);

                  },

                  child: const Text('Add'),

                  ),

             ],

            );

        },

      );

    }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Theme.of(context).colorScheme.inversePrimary,

        title: Text("Todo App"),

        centerTitle: true,

      ),

      body: ListView(

          padding: const EdgeInsets.symmetric(vertical: 8.0),

          children: \_todos.map((Todo todo) {

              return TodoItem(

                  todo: todo,

                  onTodoChanged: \_handleTodoChange,

                  removeTodo:\_deleteTodo

                );

          }).toList(),

      ),

      floatingActionButton: FloatingActionButton(

        onPressed: () => \_displayDialog(),

        tooltip: 'Add a Todo',

        child: const Icon(Icons.add),

      ),

    );

  }

}

class Todo {

    String name;

    bool completed;

    Todo({required this.name, required this.completed});

}

class TodoItem extends StatelessWidget {

  const TodoItem({

    super.key,

    required this.todo,

    required this.onTodoChanged,

    required this.removeTodo,

  });

  final Todo todo;

  final void Function(Todo todo) onTodoChanged;

  final void Function(Todo todo) removeTodo;

  TextStyle? \_getTextStyle(bool checked) {

    if (!checked) return null;

    return const TextStyle(

      color: Colors.black54,

      decoration: TextDecoration.lineThrough,

    );

  }

  @override

  Widget build(BuildContext context) {

    return ListTile(

      onTap: () => onTodoChanged(todo),

      leading: Checkbox(

        checkColor: Colors.greenAccent,

        activeColor: Colors.red,

        value: todo.completed,

        onChanged: (\_) => onTodoChanged(todo),

      ),

      title: Row(

        children: <Widget>[

          Expanded(

            child: Text(

              todo.name,

              style: \_getTextStyle(todo.completed),

            ),

          ),

          IconButton(

            iconSize: 30,

            icon: const Icon(Icons.delete, color: Colors.red),

            onPressed: () => removeTodo(todo),

          ),

        ],

      ),

    );

  }

}

**Global State Management:**

Global state, is state that needs to be accessible from multiple parts of the app. For example, if you have a shopping cart that needs to be accessed from various screens, you’ll need a way to manage that state globally.

Managing global state can become complex, especially as your app grows. This is where tools like Provider come in handy, making it easier to share and manage state across the entire app.

**State Management with Provider:**

Provider is a package that simplifies state management by allowing you to share data across your app and rebuild UI parts when the data changes. It’s perfect for beginners because it’s both easy to use and powerful enough to handle more complex cases as your app scales.

**Core Concepts**

**ChangeNotifier:** This class helps notify listeners (widgets) when the state changes. It’s the core of most state management solutions in Flutter.

**Provider:** This widget is used to make an instance of ChangeNotifier available throughout the widget tree.

**Consumer:** A widget that listens to Provider and rebuilds whenever the state it depends on changes.

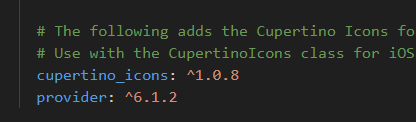
**Creating an App with Global State Management**

Let’s build an example of Counter, and this state will be managed globally using Provider. You’ll need to set up the built-in counter project and include the Provider package. Let run command in flutter terminal

flutter pub add provider

flutter pub get

After these command provider package in included in pubspec.yaml. You can check your file.



**Step 1:** Firstly, we remove comments from the code and import library of provider at top of code. Convert **MyHomePage** class into stateless widget. Now manage state with **Provider**.

import 'package:provider/provider.dart';

**Step 2:** Now, we create a class of **ChangeNotifier** class named **counterProvider.** This class extends with **ChangeNotifer** widget. Remove **incrementCounter** function **and counter** variable from the MyHomePage class and add into **counterProvider** class.

class counterProvider extends ChangeNotifier {

  int \_counter = 0;

  void incrementCounter() {

    \_counter++;

    notifyListeners();

  }

}

Step 3: In **counterProvider** class **\_counter** is private variable. We can’t access this variable in outside this class. So, create a public getter function to access from outside the class.

class counterProvider extends ChangeNotifier {

  int \_counter = 0;

  int getCounter() => \_counter;

  void incrementCounter() {

    \_counter++;

    notifyListeners();

  }

}

So if we can this **getCounter()** to get the counter outside this class.

**Step 3:** Now we user **ChangeNotifierProvider** to create an instance of **ChangeNotifier** available throughout the widget tree. We add **ChangeNotifierProvider** in **MyApp** class at home widget.

class MyApp extends StatelessWidget {

  const MyApp({super.key});

  // This widget is the root of your application.

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'Flutter Demo',

      theme: ThemeData(

        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),

        useMaterial3: true,

      ),

      home: ChangeNotifierProvider(

        create: (\_)=>counterProvider(),

        child: MyHomePage()),

    );

  }

}

Step 4: Now add Consumer widget in MyHomePage class at **home** widget. Consumer widget that listens to Provider and rebuilds whenever the state it depends on changes.

body: Center(

        child: Consumer(

          builder: (context, \_, \_\_) {

            return Column(

            mainAxisAlignment: MainAxisAlignment.center,

            children: <Widget>[

              const Text(

                'You have pushed the button this many times:',

              ),

              Text(

                '',

                style: Theme.of(context).textTheme.headlineMedium,

              ),

            ],

          );

          }

        ),

      ),

**Step 5:** Now need to reflect updated counter value on UI. We need to update this code in Text widget.

 Text('${Provider.of<counterProvider>(context,listen: true).getCounter()}',

                style: Theme.of(context).textTheme.headlineMedium,

              ),

In above code, we need identify to that text is coming Provider class.

**Step 6:** Now we need update the code in floating button widget.

     floatingActionButton: FloatingActionButton(

        onPressed: () {

          Provider.of<counterProvider>(context, listen: false)

              .incrementCounter();

        },

        tooltip: 'Increment',

        child: const Icon(Icons.add),

      ),

**Complete Code of Counter**

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

void main() {

  runApp(const MyApp());

}

class MyApp extends StatelessWidget {

  const MyApp({super.key});

  // This widget is the root of your application.

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      debugShowCheckedModeBanner: false,

      title: 'Flutter Demo',

      theme: ThemeData(

        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),

        useMaterial3: true,

      ),

      home: ChangeNotifierProvider(

          create: (\_) => counterProvider(), child: MyHomePage()),

    );

  }

}

class MyHomePage extends StatelessWidget {

  const MyHomePage({super.key});

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Theme.of(context).colorScheme.inversePrimary,

        title: Text("Counter App"),

        centerTitle: true,

      ),

      body: Center(

        child: Consumer(builder: (context, \_, \_\_) {

          return Column(

            mainAxisAlignment: MainAxisAlignment.center,

            children: <Widget>[

              const Text(

                'You have pushed the button this many times:',

              ),

              Text(

                '${Provider.of<counterProvider>(context, listen: true).getCounter()}',

                style: Theme.of(context).textTheme.headlineMedium,

              ),

            ],

          );

        }),

      ),

      floatingActionButton: FloatingActionButton(

        onPressed: () {

          Provider.of<counterProvider>(context, listen: false)

              .incrementCounter();

        },

        tooltip: 'Increment',

        child: const Icon(Icons.add),

      ),

    );

  }

}

class counterProvider extends ChangeNotifier {

  int \_counter = 0;

  int getCounter() => \_counter;

  void incrementCounter() {

    \_counter++;

    notifyListeners();

  }

}

**Assessment:**

**Lab Task 1: Build a Shopping List App Using Flutter**

**Objective**

Create a simple Shopping List app where users can:

* Add items to the list
* Mark items as purchased
* Delete items from the list

**Requirements**

1. Use StatefulWidget to manage the list.
2. Each item should have a name and a purchased status.
3. Use a TextField in a dialog to add new items.
4. Display items in a ListView.
5. Tap an item or use a checkbox to mark it as purchased (with strikethrough).
6. Include a delete icon to remove items.
7. Use a FloatingActionButton to add items.

**Lab Task 2: Answer the following questions.**

Q1: What is the purpose of the Provider package in Flutter?

Q2: What is the difference between ChangeNotifierProvider and Provider in Flutter?

Q3: How do you update and listen to changes in state using Provider?

Q4: Explain the role of Consumer in the Provider package.