UIT UNIVERSITY

CSC-318 – Mobile Application Development

LAB 8: Flutter Backend-I

Name : _____

Roll No : _____

Section : _____

Semester : _____

FALL 2024

COMPUTER SCIENCE DEPARTMENT

LAB-8 | Flutter Backend-I

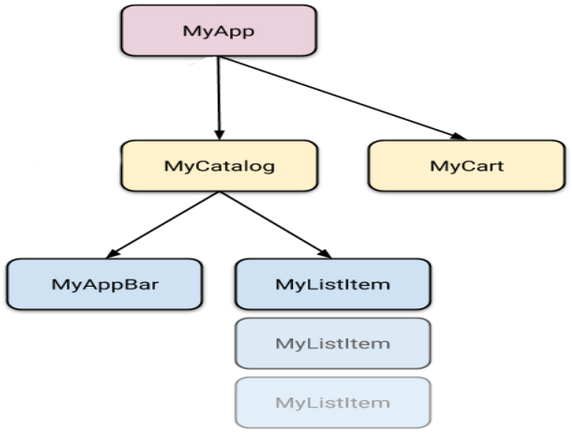
Simple app state management

An animated gif showing a Flutter app in use. It starts with the user on a login screen. They log in and are taken to the catalog screen, with a list of items. The click on several items, and as they do so, the items are marked as "added". The user clicks on a button and gets taken to the cart view. They see the items there. They go back to the catalog, and the items they bought still show "added". End of animation.
For illustration, consider the following simple app.

The app has two separate screens: a catalog, and a cart (represented by the MyCatalog, and MyCart widgets, respectively). It could be a shopping app, but you can imagine the same structure in a simple social networking app (replace catalog for "wall" and cart for "favorites").

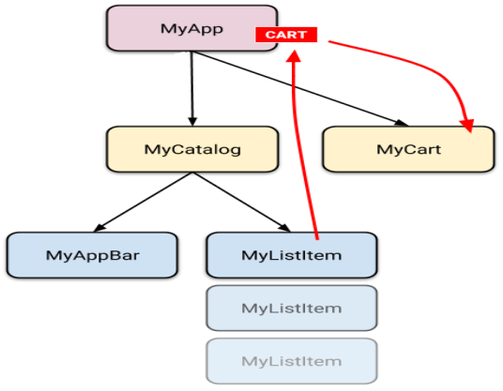
The catalog screen includes a custom app bar (MyAppBar) and a scrolling view of many list items (MyListItems).

Here's the app visualized as a widget tree



.

A widget tree with MyApp at the top, and MyCatalog and MyCart below it. MyCart area leaf nodes, but MyCatalog have two children: MyAppBar and a list of MyListItems.



Accessing the state
#
When a user clicks on one of the items in the catalog, it's added to the cart. But since the cart lives above MyListItem, how do we do that?

A simple option is to provide a callback that MyListItem can call when it is clicked. Dart's functions are first class objects, so you can pass them around any way you want. So, inside MyCatalog you can define the following:

```dart
@override
Widget build(BuildContext context) {
  return SomeWidget(
    // Construct the widget, passing it a reference to the method above.
    MyListItem(myTapCallback),
  );
}

void myTapCallback(Item item) {
  print('user tapped on $item');
}
```

This works okay, but for an app state that you need to modify from many different places, you'd have to pass around a lot of callbacks—which gets old pretty quickly.

Fortunately, Flutter has mechanisms for widgets to provide data and services to their descendants (in other words, not just their children, but any widgets below them). As you would expect from Flutter, where *Everything is a Widget™*, these mechanisms are just special kinds of widgets—InheritedWidget, InheritedNotifier, InheritedModel, and more. We won't be covering those here, because they are a bit low-level for what we're trying to do.

Instead, we are going to use a package that works with the low-level widgets but is simple to use. It's called provider.

Before working with provider, don't forget to add the dependency on it to your pubspec.yaml.

To add the provider package as a dependency, run flutter pub add:

```
flutter pub add provider
```

Now you can import 'package:provider/provider.dart'; and start building.

With provider, you don't need to worry about callbacks or InheritedWidgets. But you do need to understand 3 concepts:

- ChangeNotifier
- ChangeNotifierProvider
- Consumer

ChangeNotifier
#

ChangeNotifier is a simple class included in the Flutter SDK which provides change notification to its listeners. In other words, if something is a ChangeNotifier, you can subscribe to its changes. (It is a form of Observable, for those familiar with the term.)

In provider, ChangeNotifier is one way to encapsulate your application state. For very simple apps, you get by with a single ChangeNotifier. In complex ones, you'll have several models, and therefore several ChangeNotifiers. (You don't need to use ChangeNotifier with provider at all, but it's an easy class to work with.)

In our shopping app example, we want to manage the state of the cart in a ChangeNotifier. We create a new class that extends it, like so:

```dart
class CartModel extends ChangeNotifier {
  /// Internal, private state of the cart.
  final List<Item> _items = [];

  /// An unmodifiable view of the items in the cart.
  UnmodifiableListView<Item> get items => UnmodifiableListView(_items);

  /// The current total price of all items (assuming all items cost $42).
  int get totalPrice => _items.length * 42;

  /// Adds [item] to cart. This and [removeAll] are the only ways to modify the
  /// cart from the outside.
  void add(Item item) {
    _items.add(item);
    // This call tells the widgets that are listening to this model to rebuild.
    notifyListeners();
  }

  /// Removes all items from the cart.
  void removeAll() {
    _items.clear();
    // This call tells the widgets that are listening to this model to rebuild.
    notifyListeners();
  }
}
```

The only code that is specific to ChangeNotifier is the call to notifyListeners(). Call this method any time the model changes in a way that might change your app's UI. Everything else in CartModel is the model itself and its business logic.

ChangeNotifier is part of flutter:foundation and doesn't depend on any higher-level classes in Flutter. It's easily testable (you don't even need to use widget testing for it). For example, here's a simple unit test of CartModel:

```dart
test('adding item increases total cost', () {
  final cart = CartModel();
  final startingPrice = cart.totalPrice;
  var i = 0;
  cart.addListener(() {
    expect(cart.totalPrice, greaterThan(startingPrice));
    i++;
  });
  cart.add(Item('Dash'));
  expect(i, 1);
});
```

ChangeNotifierProvider

#

ChangeNotifierProvider is the widget that provides an instance of a ChangeNotifier to its descendants. It comes from the provider package.

We already know where to put ChangeNotifierProvider: above the widgets that need to access it. In the case of CartModel, that means somewhere above both MyCart and MyCatalog.

You don't want to place ChangeNotifierProvider higher than necessary (because you don't want to pollute the scope). But in our case, the only widget that is on top of both MyCart and MyCatalog is MyApp.

```dart
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CartModel(),
      child: const MyApp(),
    ),
  );
}
```

Note that we're defining a builder that creates a new instance of CartModel. ChangeNotifierProvider is smart enough *not* to rebuild CartModel unless absolutely necessary. It also automatically calls dispose() on CartModel when the instance is no longer needed.

If you want to provide more than one class, you can use MultiProvider:

```dart
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => CartModel()),
        Provider(create: (context) => SomeOtherClass()),
      ],
      child: const MyApp(),
    ),
  );
}
```

Consumer

#

Now that CartModel is provided to widgets in our app through the ChangeNotifierProvider declaration at the top, we can start using it.

This is done through the Consumer widget.

```dart
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text('Total price: ${cart.totalPrice}');
  },
);
```

We must specify the type of the model that we want to access. In this case, we want CartModel, so we write Consumer<CartModel>. If you don't specify the generic (<CartModel>), the provider package won't be able to help you. provider is based on types, and without the type, it doesn't know what you want.

The only required argument of the Consumer widget is the builder. Builder is a function that is called whenever the ChangeNotifier changes. (In other words, when you call notifyListeners() in your model, all the builder methods of all the corresponding Consumer widgets are called.)

The builder is called with three arguments. The first one is context, which you also get in every build method.

The second argument of the builder function is the instance of the ChangeNotifier. It's what we were asking for in the first place. You can use the data in the model to define what the UI should look like at any given point.

The third argument is child, which is there for optimization. If you have a large widget subtree under your Consumer that *doesn't* change when the model changes, you can construct it once and get it through the builder.

```dart
return Consumer<CartModel>(
  builder: (context, cart, child) => Stack(
    children: [
      // Use SomeExpensiveWidget here, without rebuilding every time.
      if (child != null) child,
      Text('Total price: ${cart.totalPrice}'),
    ],
  ),
  // Build the expensive widget here.
  child: const SomeExpensiveWidget(),
);
```

It is best practice to put your Consumer widgets as deep in the tree as possible. You don't want to rebuild large portions of the UI just because some detail somewhere changed.

```dart
// DON'T DO THIS
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return HumongousWidget(
      // ...
      child: AnotherMonstrousWidget(
        // ...
        child: Text('Total price: ${cart.totalPrice}'),
      ),
    );
  },
);
```

Instead:

```dart
// DO THIS
return HumongousWidget(
  // ...
  child: AnotherMonstrousWidget(
    // ...
    child: Consumer<CartModel>(
      builder: (context, cart, child) {
        return Text('Total price: ${cart.totalPrice}');
      },
```

```
    ),
  ),
);
```

Provider.of
#
Sometimes, you don't really need the *data* in the model to change the UI but you still need to access it. For example, a ClearCart button wants to allow the user to remove everything from the cart. It doesn't need to display the contents of the cart, it just needs to call the clear() method.

We could use Consumer<CartModel> for this, but that would be wasteful. We'd be asking the framework to rebuild a widget that doesn't need to be rebuilt.

For this use case, we can use Provider.of, with the listen parameter set to false.

```dart
Provider.of<CartModel>(context, listen: false).removeAll();
```

Using the above line in a build method won't cause this widget to rebuild when notifyListeners is called.

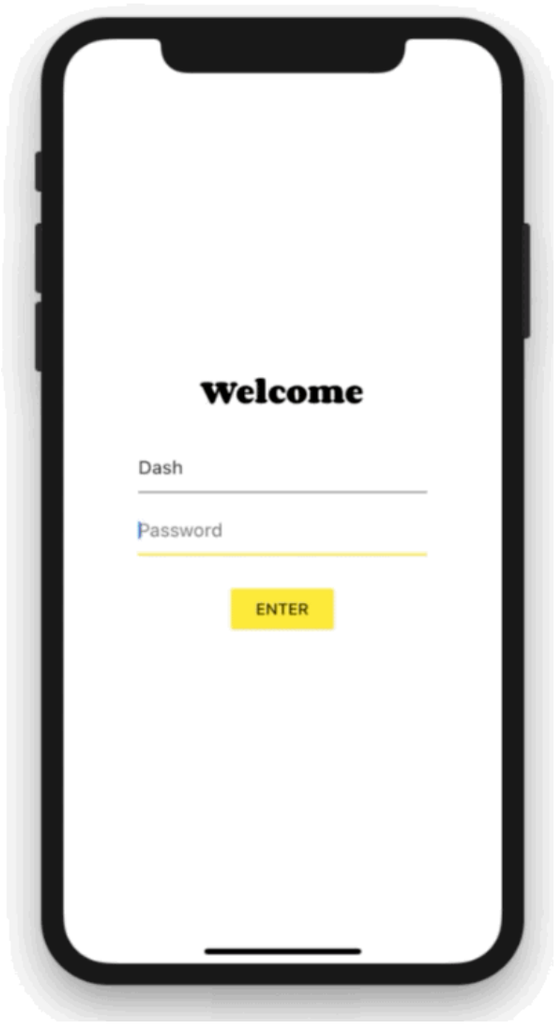
Putting it all together

Lib/screens/main.dart

```
//import 'dart:io' show Platform;

//import 'package:flutter/foundation.dart' show kIsWeb;
import 'package:flutter/material.dart';
import 'package:go_router/go_router.dart';
import 'package:provider/provider.dart';
import 'package:flutter_application_2/common/theme.dart';
import 'package:flutter_application_2/models/cart.dart';
import 'package:flutter_application_2/models/catalog.dart';
import 'package:flutter_application_2/screens/cart.dart';
import 'package:flutter_application_2/screens/catalog.dart';
import 'package:flutter_application_2/screens/login.dart';
//import 'package:window_size/window_size.dart';

void main() {

  runApp(const MyApp());
}

const double windowWidth = 400;
const double windowHeight = 800;

GoRouter router() {
  return GoRouter(
    initialLocation: '/login',
    routes: [
```

```
      GoRoute(
        path: '/login',
        builder: (context, state) => const MyLogin(),
      ),
      GoRoute(
        path: '/catalog',
        builder: (context, state) => const MyCatalog(),
        routes: [
          GoRoute(
            path: 'cart',
            builder: (context, state) => const MyCart(),
          ),
        ],
      ),
    ],
  );
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    // Using MultiProvider is convenient when providing multiple objects.
    return MultiProvider(
      providers: [
        // In this sample app, CatalogModel never changes, so a simple Provider
        // is sufficient.
        Provider(create: (context) => CatalogModel()),
        // CartModel is implemented as a ChangeNotifier, which calls for the use
        // of ChangeNotifierProvider. Moreover, CartModel depends
        // on CatalogModel, so a ProxyProvider is needed.
        ChangeNotifierProxyProvider<CatalogModel, CartModel>(
          create: (context) => CartModel(),
          update: (context, catalog, cart) {
            if (cart == null) throw ArgumentError.notNull('cart');
            cart.catalog = catalog;
            return cart;
          },
        ),
      ],
      child: MaterialApp.router(
        title: 'Provider Demo',
        theme: appTheme,
        routerConfig: router(),
      ),
    );
  }
```

}



Lib/screens/login.dart

```
// Copyright 2020 The Flutter team. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

import 'package:flutter/material.dart';
import 'package:go_router/go_router.dart';

class MyLogin extends StatelessWidget {
  const MyLogin({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
```

```
        child: Container(
          padding: const EdgeInsets.all(80.0),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Text(
                'Welcome',
                style: Theme.of(context).textTheme.displayLarge,
              ),
              TextFormField(
                decoration: const InputDecoration(
                  hintText: 'Username',
                ),
              ),
              TextFormField(
                decoration: const InputDecoration(
                  hintText: 'Password',
                ),
                obscureText: true,
              ),
              const SizedBox(
                height: 24,
              ),
              ElevatedButton(
                onPressed: () {
                  context.pushReplacement('/catalog');
                },
                style: ElevatedButton.styleFrom(
                  backgroundColor: Colors.yellow,
                ),
                child: const Text('ENTER'),
              )
            ],
          ),
        ),
      ),
    );
  }
}
```

Lib/screens/catalog.dart

```
// Copyright 2019 The Flutter team. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

import 'package:flutter/material.dart';
import 'package:go_router/go_router.dart';
```

```
import 'package:provider/provider.dart';
import 'package:flutter_application_2/models/cart.dart';
import 'package:flutter_application_2/models/catalog.dart';

class MyCatalog extends StatelessWidget {
  const MyCatalog({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: CustomScrollView(
        slivers: [
          _MyAppBar(),
          const SliverToBoxAdapter(child: SizedBox(height: 12)),
          SliverList(
            delegate: SliverChildBuilderDelegate(
                (context, index) => _MyListItem(index)),
          ),
        ],
      ),
    );
  }
}

class _AddButton extends StatelessWidget {
  final Item item;

  const _AddButton({required this.item});

  @override
  Widget build(BuildContext context) {
    // The context.select() method will let you listen to changes to
    // a *part* of a model. You define a function that "selects" (i.e. returns)
    // the part you're interested in, and the provider package will not rebuild
    // this widget unless that particular part of the model changes.
    //
    // This can lead to significant performance improvements.
    var isInCart = context.select<CartModel, bool>(
      // Here, we are only interested whether [item] is inside the cart.
      (cart) => cart.items.contains(item),
    );

    return TextButton(
      onPressed: isInCart
          ? null
          : () {
              // If the item is not in cart, we let the user add it.
              // We are using context.read() here because the callback
```

```
            // is executed whenever the user taps the button. In other
            // words, it is executed outside the build method.
            var cart = context.read<CartModel>();
            cart.add(item);
          },
      style: ButtonStyle(
        overlayColor: WidgetStateProperty.resolveWith<Color?>((states) {
          if (states.contains(WidgetState.pressed)) {
            return Theme.of(context).primaryColor;
          }
          return null; // Defer to the widget's default.
        }),
      ),
      child: isInCart
          ? const Icon(Icons.check, semanticLabel: 'ADDED')
          : const Text('ADD'),
    );
  }
}

class _MyAppBar extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return SliverAppBar(
      title: Text('Catalog', style: Theme.of(context).textTheme.displayLarge),
      floating: true,
      actions: [
        IconButton(
          icon: const Icon(Icons.shopping_cart),
          onPressed: () => context.go('/catalog/cart'),
        ),
      ],
    );
  }
}

class _MyListItem extends StatelessWidget {
  final int index;

  const _MyListItem(this.index);

  @override
  Widget build(BuildContext context) {
    var item = context.select<CatalogModel, Item>(
      // Here, we are only interested in the item at [index]. We don't care
      // about any other change.
      (catalog) => catalog.getByPosition(index),
    );
```

```
    var textTheme = Theme.of(context).textTheme.titleLarge;

    return Padding(
      padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 8),
      child: LimitedBox(
        maxHeight: 48,
        child: Row(
          children: [
            AspectRatio(
              aspectRatio: 1,
              child: Container(
                color: item.color,
              ),
            ),
            const SizedBox(width: 24),
            Expanded(
              child: Text(item.name, style: textTheme),
            ),
            const SizedBox(width: 24),
            _AddButton(item: item),
          ],
        ),
      ),
    );
  }
}
```

Lib/screens/cart.dart

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'package:flutter_application_2/models/cart.dart';

class MyCart extends StatelessWidget {
  const MyCart({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Cart', style: Theme.of(context).textTheme.displayLarge),
        backgroundColor: Colors.white,
      ),
      body: Container(
        color: Colors.yellow,
        child: Column(
          children: [
            Expanded(
              child: Padding(
```

```
              padding: const EdgeInsets.all(32),
              child: _CartList(),
            ),
          ),
          const Divider(height: 4, color: Colors.black),
          _CartTotal()
        ],
      ),
    ),
  );
 }
}

class _CartList extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  var itemNameStyle = Theme.of(context).textTheme.titleLarge;
  // This gets the current state of CartModel and also tells Flutter
  // to rebuild this widget when CartModel notifies listeners (in other words,
  // when it changes).
  var cart = context.watch<CartModel>();

  return ListView.builder(
   itemCount: cart.items.length,
   itemBuilder: (context, index) => ListTile(
    leading: const Icon(Icons.done),
    trailing: IconButton(
     icon: const Icon(Icons.remove_circle_outline),
     onPressed: () {
      cart.remove(cart.items[index]);
     },
    ),
    title: Text(
     cart.items[index].name,
     style: itemNameStyle,
    ),
   ),
  );
 }
}

class _CartTotal extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  var hugeStyle =
    Theme.of(context).textTheme.displayLarge!.copyWith(fontSize: 48);

  return SizedBox(
```

```
      height: 200,
      child: Center(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            // Another way to listen to a model's change is to include
            // the Consumer widget. This widget will automatically listen
            // to CartModel and rerun its builder on every change.
            //
            // The important thing is that it will not rebuild
            // the rest of the widgets in this build method.
            Consumer<CartModel>(
                builder: (context, cart, child) =>
                    Text('\$${cart.totalPrice}', style: hugeStyle)),
            const SizedBox(width: 24),
            FilledButton(
              onPressed: () {
                ScaffoldMessenger.of(context).showSnackBar(
                    const SnackBar(content: Text('Buying not supported yet.')));
              },
              style: TextButton.styleFrom(foregroundColor: Colors.white),
              child: const Text('BUY'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Lib/models/catalog.dart

```
import 'package:flutter/material.dart';

/// A proxy of the catalog of items the user can buy.
///
/// In a real app, this might be backed by a backend and cached on device.
/// In this sample app, the catalog is procedurally generated and infinite.
///
/// For simplicity, the catalog is expected to be immutable (no products are
/// expected to be added, removed or changed during the execution of the app).

class CatalogModel {
  static List<String> itemNames = [
    'Code Smell',
    'Control Flow',
    'Interpreter',
    'Recursion',
    'Sprint',
```

```
    'Heisenbug',
    'Spaghetti',
    'Hydra Code',
    'Off-By-One',
    'Scope',
    'Callback',
    'Closure',
    'Automata',
    'Bit Shift',
    'Currying',
  ];

  /// Get item by [id].
  ///
  /// In this sample, the catalog is infinite, looping over [itemNames].
  Item getById(int id) => Item(id, itemNames[id % itemNames.length]);

  /// Get item by its position in the catalog.
  Item getByPosition(int position) {
    // In this simplified case, an item's position in the catalog
    // is also its id.
    return getById(position);
  }
}

@immutable
class Item {
  final int id;
  final String name;
  final Color color;
  final int price = 42;

  Item(this.id, this.name)
      // To make the sample app look nicer, each item is given one of the
      // Material Design primary colors.
      : color = Colors.primaries[id % Colors.primaries.length];

  @override
  int get hashCode => id;

  @override
  bool operator ==(Object other) => other is Item && other.id == id;
}
```

Lib/model/cart.dart

```
// Copyright 2019 The Flutter team. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.
```

```
import 'package:flutter/foundation.dart';
import 'package:flutter_application_2/models/catalog.dart';

class CartModel extends ChangeNotifier {
  /// The private field backing [catalog].
  late CatalogModel _catalog;

  /// Internal, private state of the cart. Stores the ids of each item.
  final List<int> _itemIds = [];

  /// The current catalog. Used to construct items from numeric ids.
  CatalogModel get catalog => _catalog;

  set catalog(CatalogModel newCatalog) {
    _catalog = newCatalog;
    // Notify listeners, in case the new catalog provides information
    // different from the previous one. For example, availability of an item
    // might have changed.
    notifyListeners();
  }

  /// List of items in the cart.
  List<Item> get items => _itemIds.map((id) => _catalog.getById(id)).toList();

  /// The current total price of all items.
  int get totalPrice =>
      items.fold(0, (total, current) => total + current.price);

  /// Adds [item] to cart. This is the only way to modify the cart from outside.
  void add(Item item) {
    _itemIds.add(item.id);
    // This line tells [Model] that it should rebuild the widgets that
    // depend on it.
    notifyListeners();
  }

  void remove(Item item) {
    _itemIds.remove(item.id);
    // Don't forget to tell dependent widgets to rebuild _every time_
    // you change the model.
    notifyListeners();
  }
}
```

Lib/common/theme.dart

```
import 'package:flutter/material.dart';
```

```
final appTheme = ThemeData(
  colorSchemeSeed: Colors.yellow,
  textTheme: const TextTheme(
    displayLarge: TextStyle(
      fontFamily: 'Corben',
      fontWeight: FontWeight.w700,
      fontSize: 24,
      color: Colors.black,
    ),
  ),
);
```

Task-01:

Event Management Application

Project Overview:

This app will allow users to:

1. **Browse Events**: View a list of upcoming events.
2. **Event Details**: View the details of each event.
3. **Register for an Event**: Users can register for events they are interested in.
4. **View Registrations**: Users can see the events they have registered for.
5. **Event Search and Filters**: Users can search for events by name and filter by categories (e.g., music, technology, sports).

Task Breakdown:

---

1. Event List Screen

- **Description**: The first screen users will see is a list of upcoming events.
- **Goal**:
  - Display a list of events, each with a title, date, and category.
  - When tapped, the event should navigate to a **Event Details Screen**.

---

2. Event Details Screen

- **Description**: This screen displays detailed information about a selected event.
- **Goal**:
  - Display the full details of the event (name, date, description, category, and a registration button).
  - Add a button for **Registering** for the event.

- If the user has already registered, show a "Already Registered" message.

---

3. Register for Event

- **Description**: This feature allows users to register for events they are interested in.
- **Goal**:
  - When the user taps "Register" in the **Event Details Screen**, they should be added to the event's registration list.
  - The event should show a confirmation message, and the button should change to "Already Registered" once the user registers.

---

4. View Registered Events

- **Description**: The user can view all the events they've registered for.
- **Goal**:
  - Create a **Registered Events Screen** that lists all events the user has registered for.
  - The list should show event names and dates.

---

5. Search and Filter Events

- **Description**: Allow the user to search and filter events by name or category.
- **Goal**:
  - Add a **Search Bar** at the top of the **Event List Screen** to allow users to search by event name.
  - Add a **Filter Option** (such as a dropdown or list) to allow users to filter events by categories like "Music", "Technology", "Sports", etc.

---

6. Implement Custom Themes

- **Description**: Add a light/dark mode toggle for the app.
- **Goal**:
  - Add a **Theme Toggle** button in the **AppBar** so users can switch between light and dark themes.
  - Customize the app's theme to have colors suitable for both light and dark modes.

---

7. Enhance the UI with Design Patterns

- **Description**: Improve the user interface with design patterns that make the app look polished and professional.
- **Goal**:
  - Use `Cards`, `ListTile`, and `Image` widgets to display event information.
  - Ensure a smooth user experience by making the registration flow simple and clear.

Updated Project Structure:

1. **lib/screens/event_list.dart**: Displays a list of all upcoming events.
2. **lib/screens/event_details.dart**: Shows the details of a selected event with a registration button.
3. **lib/screens/registered_events.dart**: Displays a list of events the user has registered for.
4. **lib/models/event.dart**: Contains the data structure for events (name, date, category, description).
5. **lib/models/registration.dart**: Handles the logic for managing user registrations for events.
6. **lib/common/theme.dart**: Defines the app's light and dark themes.

Task Steps:

Step 1: Create Event List Screen

- **Description**: The `EventListScreen` should display a list of events.
- **Goal**:
  - Create a screen that lists all upcoming events.
  - Each event in the list should show a title, date, and category.
  - When clicked, the event should navigate to the **EventDetailsScreen**.

Step 2: Create Event Details Screen

- **Description**: This screen will show detailed information about the event the user selects.
- **Goal**:
  - Include details like event name, description, date, and category.
  - Include a **Register** button that registers the user for the event.
  - If the user is already registered, display a message saying "You are already registered for this event."

Step 3: Implement Event Registration Logic

- **Description**: Implement the logic that allows the user to register for events.
- **Goal**:
  - Use a state management solution like `Provider` to manage the user's event registrations.
  - When the user presses the **Register** button, the event should be added to their list of registered events.

Step 4: Create Registered Events Screen

- **Description**: This screen should list all the events the user has registered for.
- **Goal**:
  - Use the state management solution to retrieve and display the user's registered events.
  - Display the name and date of each event the user is registered for.

Step 5: Add Event Search and Filters

- **Description**: Add functionality to search for events by name and filter events by category.
- **Goal**:
  - Add a **Search Bar** at the top of the `EventListScreen`.
  - Add a dropdown or list to allow users to filter events by category.

Step 6: Implement Theme Toggle

- **Description**: Add a button in the app's `AppBar` to toggle between light and dark modes.
- **Goal**:
  - Define two themes in `theme.dart` (light and dark).
  - Add a toggle button in the app's main navigation to switch between the two themes.

Step 7: Polish the UI

- **Description**: Improve the design of the app by adding well-styled widgets and ensuring the user interface is clean and user-friendly.
- **Goal**:
  - Use `Card`, `ListTile`, and `Image` widgets to create a visually appealing display of events.
  - Make sure the **Register** button looks prominent, and the overall user experience feels smooth and polished.

Expected Deliverables:

1. **Event List Screen**: A list of upcoming events with the ability to navigate to event details.
2. **Event Details Screen**: A screen showing full event details with a registration button.
3. **Registered Events Screen**: A screen showing all the events the user is registered for.
4. **Event Search and Filters**: A functional search bar and category filter for events.
5. **Theme Toggle**: Ability to toggle between light and dark themes.
6. **State Management**: Functional state management that tracks user registrations and updates the UI accordingly.
7. **UI Enhancements**: Clean, well-organized UI that improves the user experience.

Bonus Challenge (Optional):

- **Task**: Add persistence for event registrations using `SharedPreferences` or `Hive` so that user registrations are saved even if the app is closed and reopened.

Submission Instructions:

- Zip the project and submit it to the course platform.

- Include a short video or screenshots demonstrating the features you have implemented (optional).

Evaluation Criteria:

- **Correctness**: The app should function as expected, with all features implemented correctly.
- **Code Quality**: Code should be clean, organized, and follow Flutter best practices.
- **UI/UX**: The app should have a visually appealing design and a smooth user experience.
- **Testing**: The app should be free of runtime errors and all features should be tested and functional.