Interacting with RESTful API

`Flutter`, `http` and `Bloc`

Learning Outcomes

After completing this session you should be able to

- Interact with **`REST API`** using **`http`** package and **`bloc`**
- Explore the Layered Architecture of Bloc
- Use Flutter forms
- Validate Flutter forms
- Organize code files based on application feature and layered architecture
- Organize imports using barrel files

Introduction

The following slides explain a sample **CRUD** application named **CourseApp** which

- Interacts with a **REST API**
- Uses **Bloc**'s Architecture

The full code of the application is linked below

https://github.com/betsegawlemma/course_flutter_app/tree/main/lib

Making Network Request

Adding the **`http`** package dependency in your **`pubspec.yaml`** file

```
Dependencies:
  flutter:
    sdk: flutter
  http: ^0.12.2
```

Making Network Request

Importing the **`http`** package

```
import 'package:http/http.dart' as http;
```

Making Network Request

Adding the Internet permission in your **`AndroidManifest.xml`** file

```
<!-- Required to fetch data from the internet. -->
<uses-permission android:name="android.permission.INTERNET" />
```

Making Network Request

We will use the model shown in the next slide to demonstrate how to perform CRUD operation on a remote REST API

The CRUD methods shown in subsequent slides are used in the **`CourseDataProvider`** class later

The base URL of the rest API is

```
baseUrl = http://localhost
```

```
class Course {
 String id;
 String title;
 String description;
 int ects;

 Course({this.id, this.title, this.description, this.ects});

 factory Course.fromJson(Map<String, dynamic> json) {
   return Course(
     id: json['id'],
     title: json['title'],
     description: json['description'],
     ects: json['ects'],
   );
 }
}
```

CRUD: Create Data

Sending `POST` request to REST API

- Use the `post` method provided by the `http` package

```
Future<Course> createCourse(Course course) async {
  final response = await http.post(
    Uri.https('localhost:3000', '/courses'),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8' ,
    },
    body: jsonEncode(<String, String>{
        'title': course.title,
        'description': course.description,
        'ects': course.ects,
    }),
  );
  if (response.statusCode == 200) {
    return Course.fromJson(jsonDecode(response.body));
  } else {
    throw Exception('Failed to load course');
  }
}
```

CRUD: Read Data

Sending `GET` request to REST API

- Use the `get` method provided by the `http` package

```
Future<Course> getCourse(String id) async {
  final response = await http.get('$baseUrl/courses/$id');

  if (response.statusCode == 200) {
    return Course.fromJson(jsonDecode(response.body));
  } else {
    throw Exception('Failed to load course');
  }
}
```

CRUD: Update Data

Sending **`UPDATE`** request to REST API

- Use the **`put`** method provided by the **`http`** package

```
Future<void> updateCourse(Course course) async {
  final http.Response response = await http.put(
    '$baseUrl/courses/${course.id}',
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, dynamic>{
      'id': course.id,
      'title': course.title,
      'description': course.description,
      'ects': course.ects,
    }),
  );
  if (response.statusCode != 204) {
     throw Exception('Failed to load course');
  }
}
```

CRUD: Delete Data

Sending **`DELETE`** request to REST API

- Use the **`delete`** method provided by the **`http`** package

```
Future<void> deleteCourse(String id) async {
  final http.Response response = await http.delete(
    '$baseUrl/courses/$id',
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
  );

  if (response.statusCode != 204) {
    throw Exception('Failed to delete course.');
  }
}
```

Sample CourseApp

Simple Flutter CRUD Application to Manage Course Info

- **`Add`** New Course
- **`Update`** Course Info
- **`Delete`** Course Info
- **`List`** All courses

CourseApp: The REST API

POST Request

```
curl -d '{"title": "AMP", "code":"SITE0132",
"description": "lorem ipsum", "ects": 7}' -H
'Content-Type: application/json' -X POST
http://localhost:3000/courses/
```

Result (returns `200 OK` on success)

```
{"id":"60266dfa3eb8d71118fb4c22","title":"AMP","code":"SI
TE0132","description":"lorem ipsum","ects":7}
```

The REST API

GET Request

```
curl -X GET http://localhost:3000/courses
```

Result (returns `200 OK` on success)


```
[{"id":"60266dfa3eb8d71118fb4c22","title":"AMP","code":"S
ITE0132","description":"lorem
ipsum","ects":7},{"id":"60266eaa3eb8d71118fb4c23","title"
:"SEII","code":"SITE0232","description":"Lorem
Ipsum","ects":5}]
```

The REST API

PUT Request

```
curl -d '{"title": "Software Engineering
II","code":"SITE0232", "description": "lorem ipsum",
"ects": 7}' -H 'Content-Type: application/json' -X PUT
http://localhost:3000/courses/60266eaa3eb8d71118fb4c23
```

Result (returns `204 No Content` on success)

No content

The REST API

`DELETE` Request

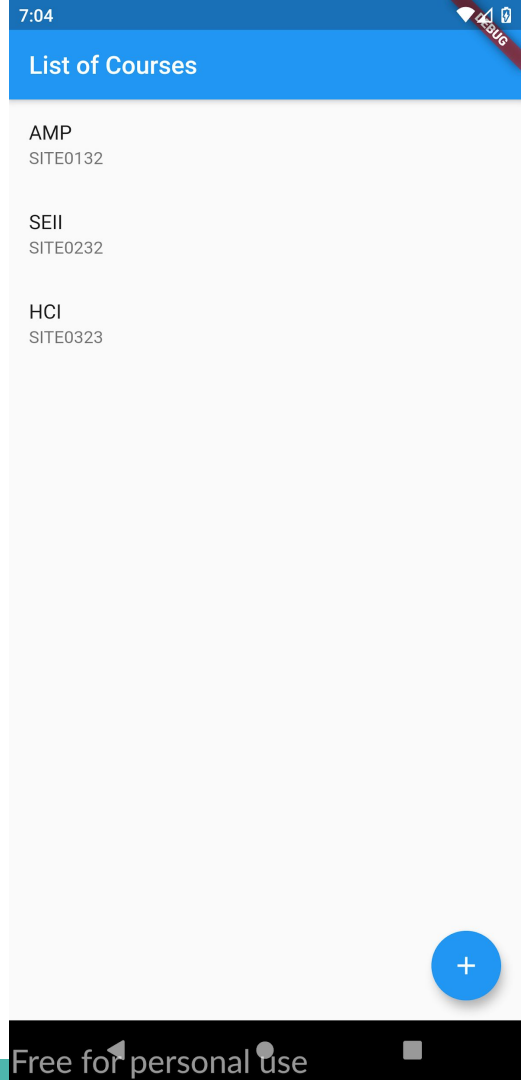
```
curl -X DELETE
http://localhost:3000/courses/60266eaa3eb8d71118fb4c23
```

Result (returns `204 No Content` on success)

No content

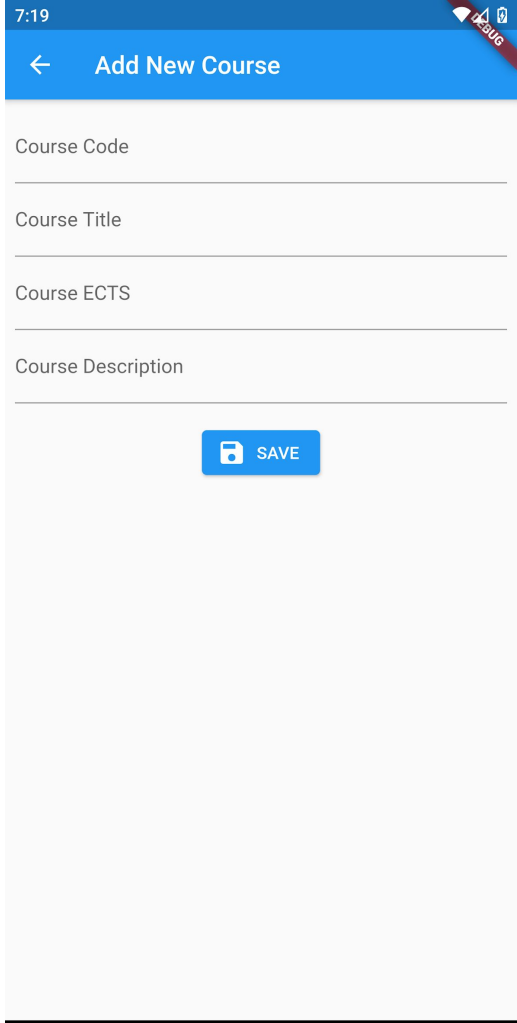
Sample CRUD Application

`List of Courses` Screen



Sample CRUD Application

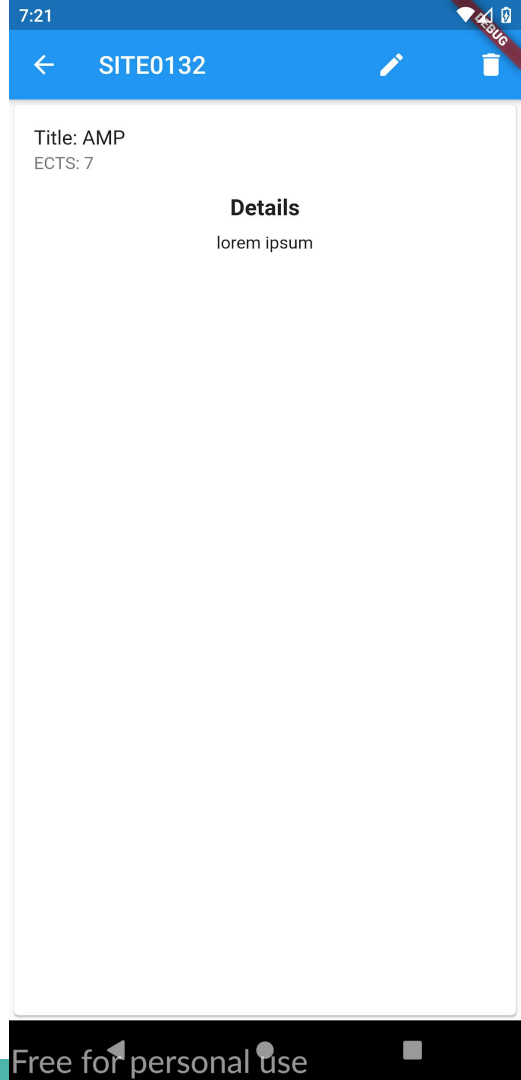
`Add New Course` Screen



Sample CRUD Application

`Course Detail` Screen

Notice the **`Edit`** and **`Delete`** icons at the right side of the `AppBar`

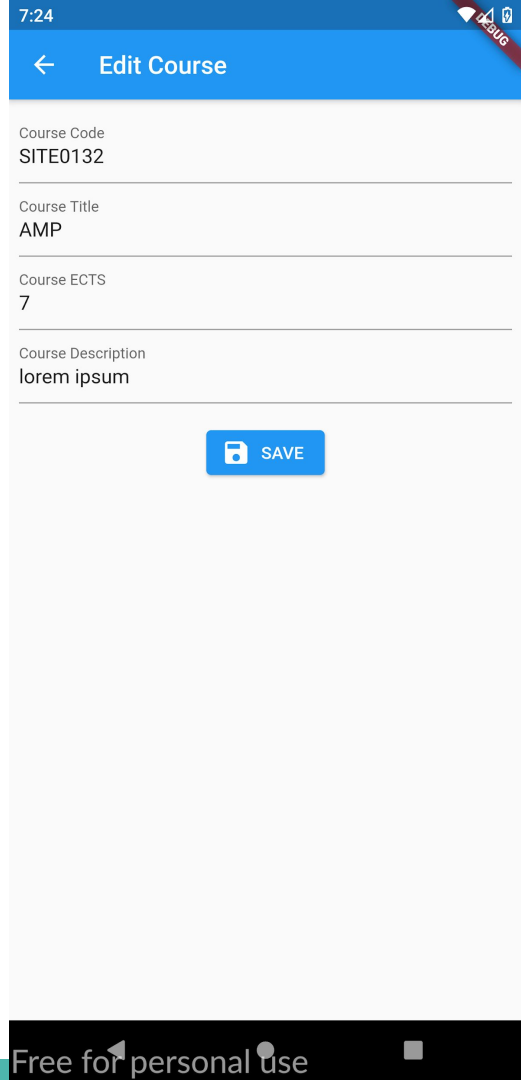


Sample CRUD Application

`Edit Course` Screen

This is the same screen as `Add Course` screen

The difference is that now the form is populated with the tobe updated course information



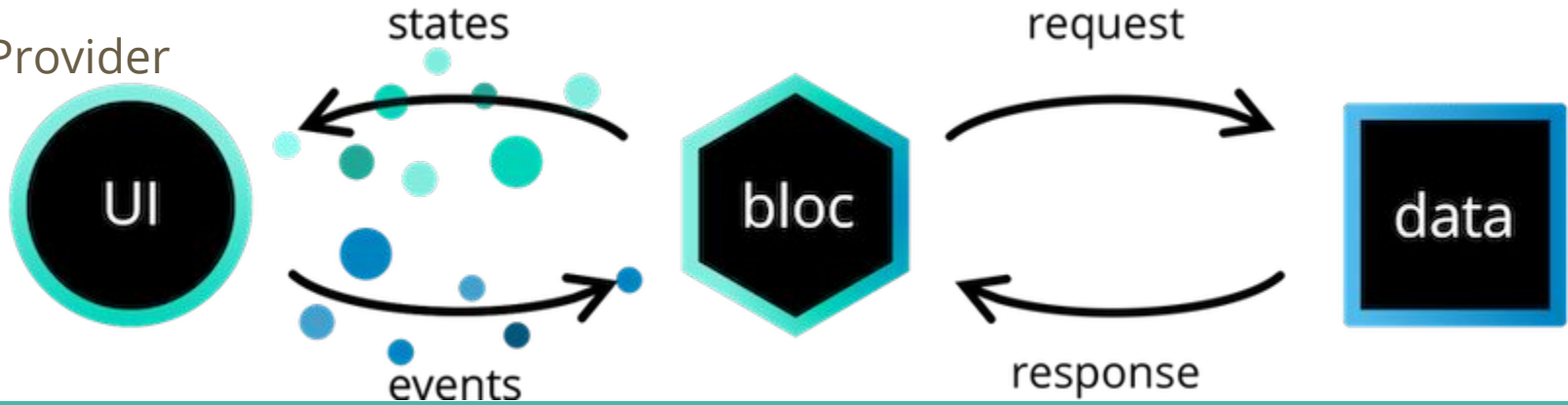
CourseApp Architecture

Presentation

Business Logic

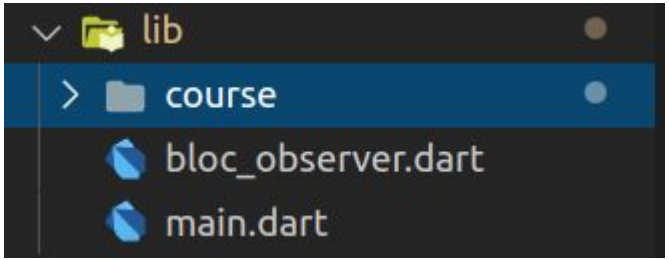
Data

- Repository
- Data Provider

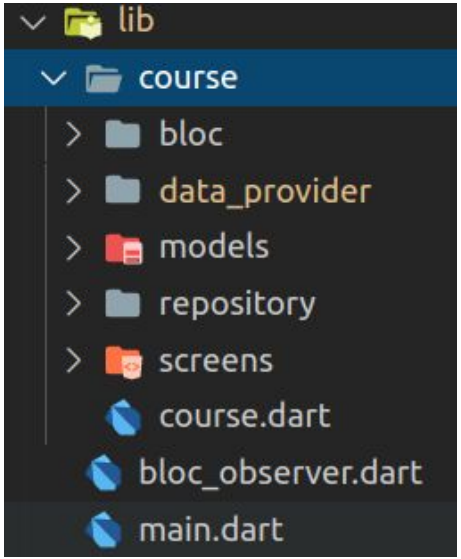


Code Structure

Create separate directory for each feature inside the `lib` directory



Inside the feature directory have separate directory for each layer



Libraries

The following additional libraries are used

- **flutter_bloc**
- **http**
- **equatable**

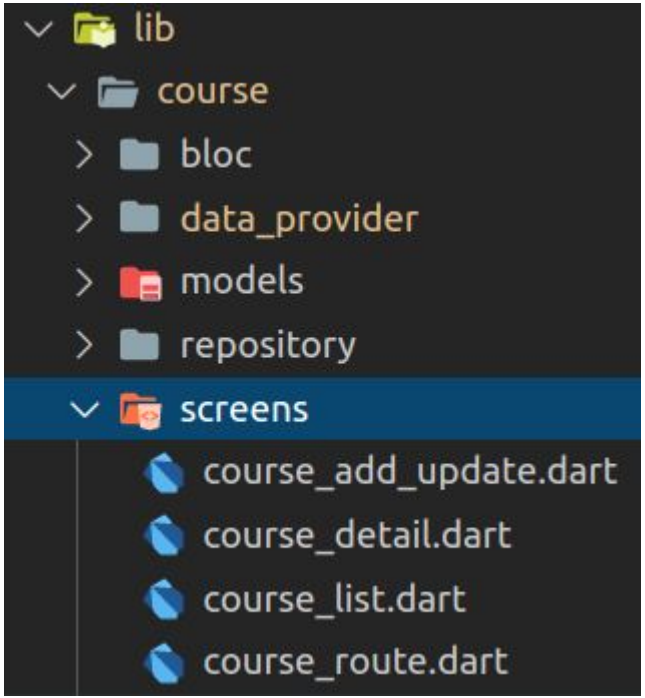
```
dependencies:
  flutter:
    sdk: flutter
  flutter_bloc: ^6.1.2
  http: ^0.12.2
  equatable: ^1.2.6
  meta: ^1.2.4
```

CourseApp: Screens

`course_add_update.dart`

Contains widgets for Adding and Updating course information

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_add_update.dart

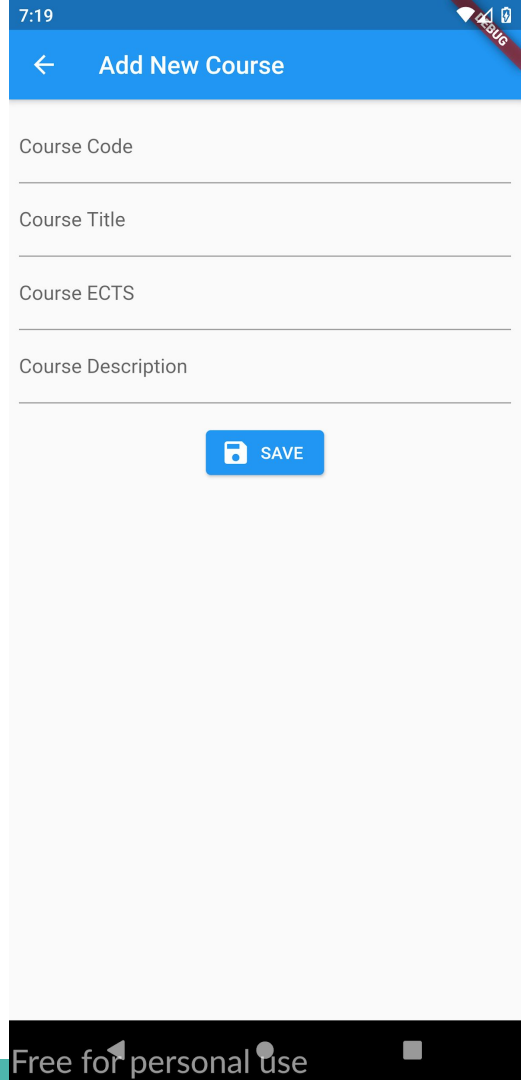


CourseApp: Screens

course_add_update.dart

Contains widgets for Adding and Updating course information

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_add_update.dart

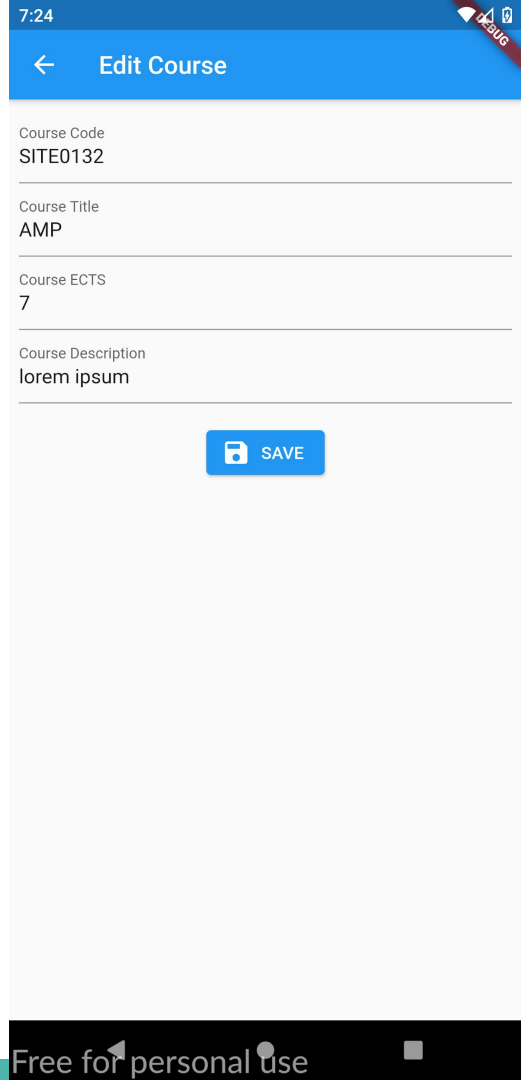


CourseApp: Screens

course_add_update.dart

Contains widgets for Adding and Updating course information

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_add_update.dart

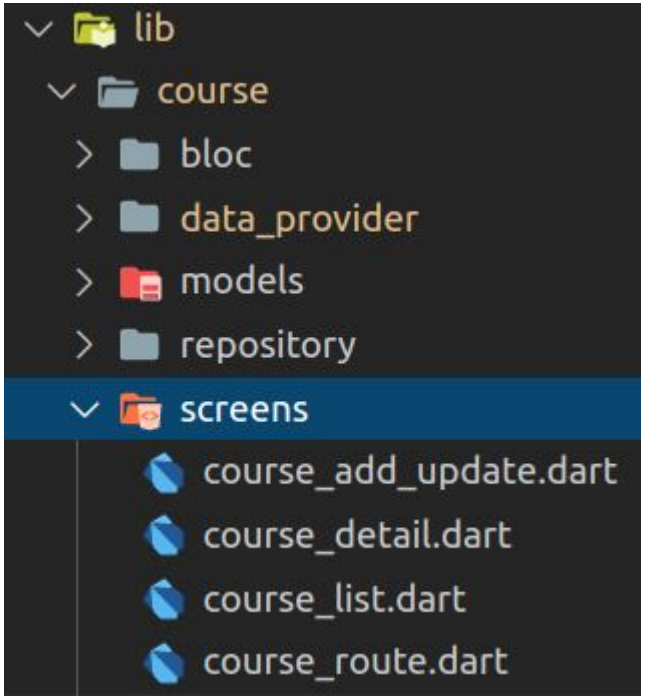


CourseApp: Screens

`course_detail.dart`

Contains widgets to display course details

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_detail.dart

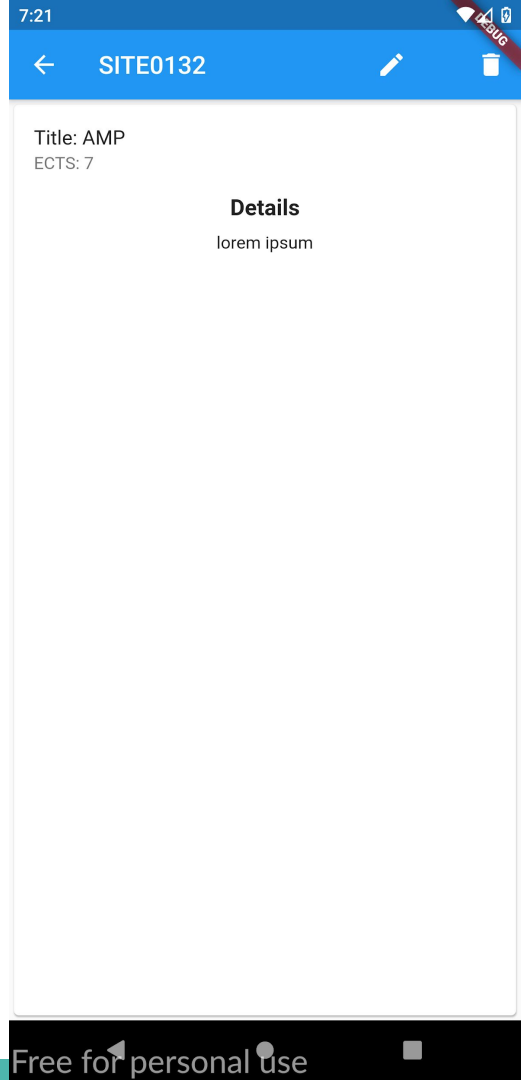


CourseApp: Screens

`course_detail.dart`

Contains widgets to display course details

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_detail.dart

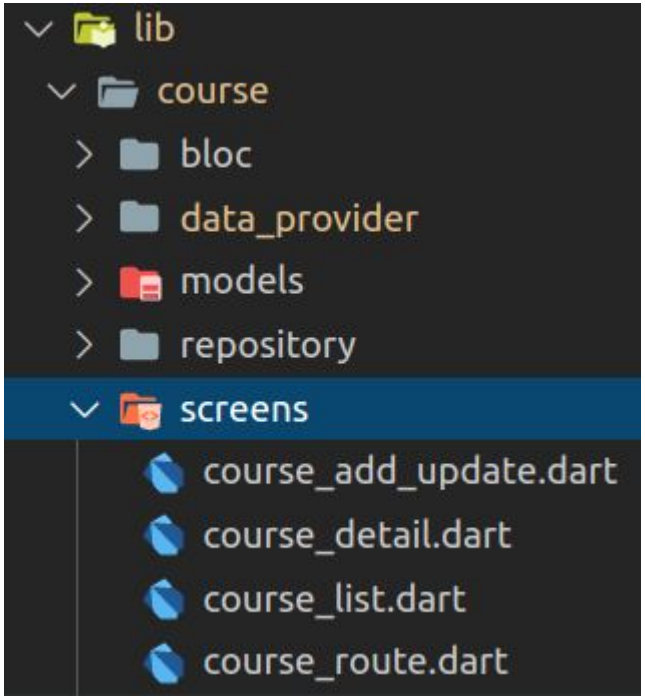


CourseApp: Screens

`course_list.dart`

Contains widgets to list course details

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_list.dart

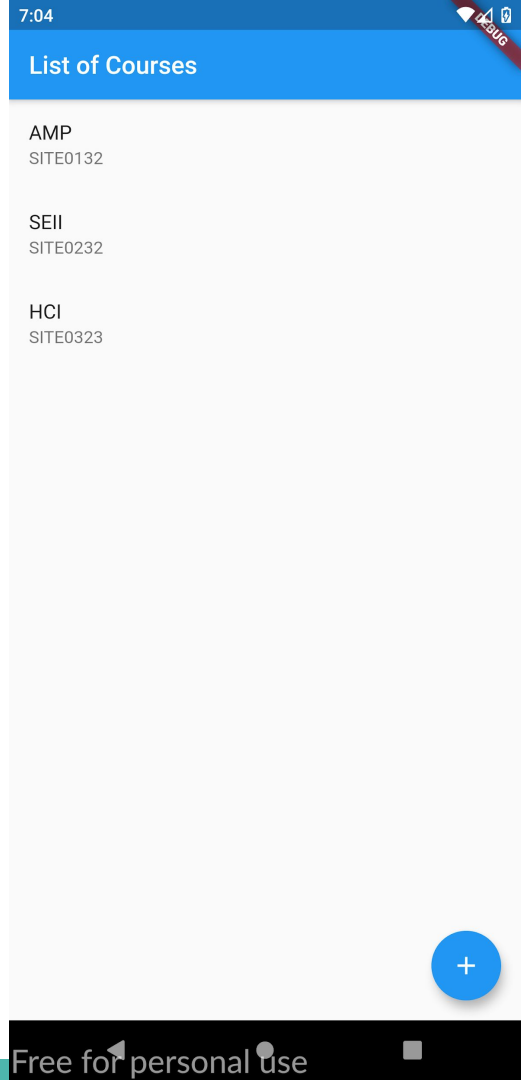


CourseApp: Screens

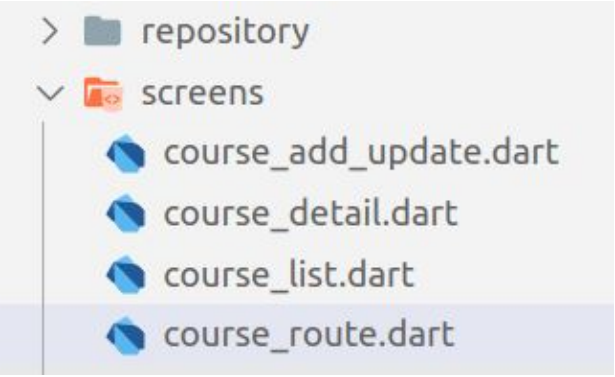
course_list.dart

Contains widgets to list course details

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_list.dart



CourseApp: Screens



`course_route.dart`

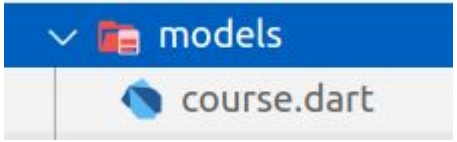
Defines Application Routes

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_route.dart

CourseApp: Course Model

Uses **`Equatable`** class for equality comparison

Has **`fromJson`** method to convert a `JSON` object returned from `REST API` request to `Course` object




```
class Course extends Equatable {
  Course(
      {this.id, @required this.title,
      @required this.code, @required this.description,
      @required this.ects});

  final String id;
  final String title;
  final String code;
  final String description;
  final int ects;

  @override
  List<Object> get props => [id, title, code, description, ects];

  factory Course.fromJson(Map<String, dynamic> json) {…

  @override
  String toString() => 'Course { id: $id, code: $code, ects: $ects }';
}
```

CourseApp: Data Provider

We have a method for each of the **CRUD** operations mentioned before

Uses **http** package to interact with the remote API (or to perform the **CRUD** requests)

```
class CourseDataProvider {
  final String _baseUrl = 'http://192.168.57.1:3000';
  final http.Client httpClient;

  CourseDataProvider({@required this.httpClient});

  Future<Course> createCourse(Course course) async {…

  Future<List<Course>> getCourses() async {…

  Future<void> deleteCourse(String id) async {…

  Future<void> updateCourse(Course course) async {…
}
```

CourseApp: Data Provider

The full code

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/data_provider/course_data.dart

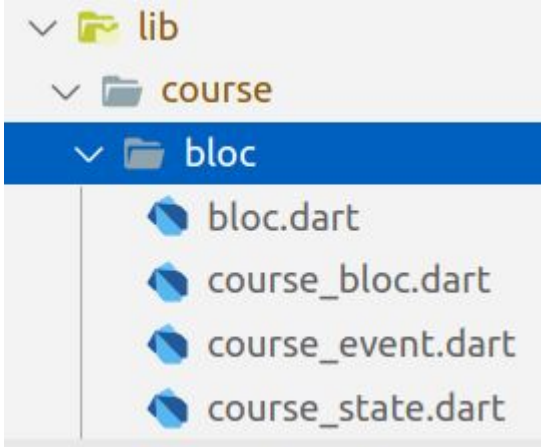
CourseApp: Repository

Just abstraction of the details of the data provider layer

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/repository/course_repository.dart

```
class CourseRepository {
  final CourseDataProvider dataProvider;

  CourseRepository({@required this.dataProvider});

  Future<Course> createCourse(Course course) async {
    return await dataProvider.createCourse(course);
  }
  Future<List<Course>> getCourses() async {
    return await dataProvider.getCourses();
  }
  Future<void> updateCourse(Course course) async {
    await dataProvider.updateCourse(course);
  }
  Future<void> deleteCourse(String id) async {
    await dataProvider.deleteCourse(id);
  }
}
```

CourseApp: Bloc



`course_event.dart`

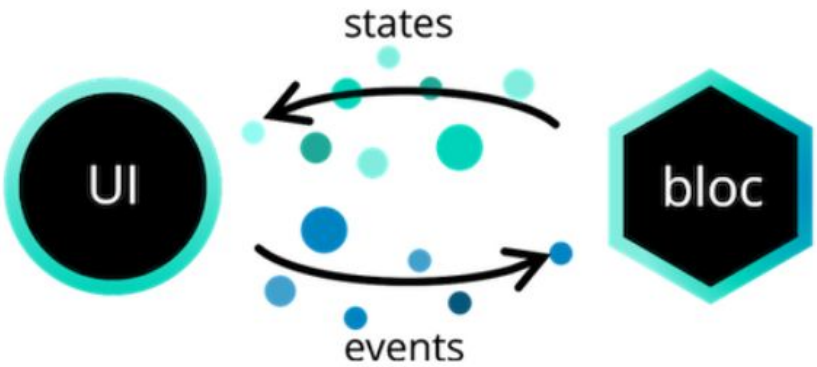
Defines the events

`course_state.dart`

Defines the states

`course_bloc.dart`

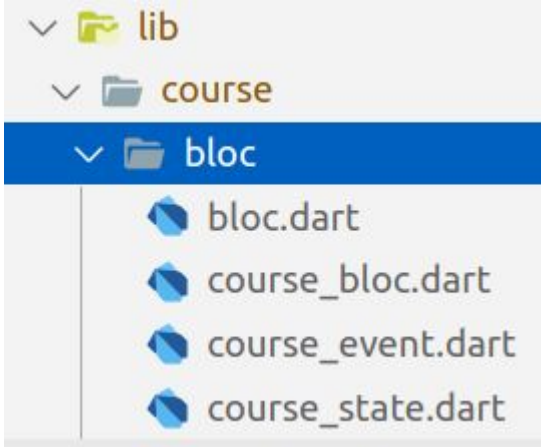
Maps Events to State



https://github.com/betsegawlemma/course_flutter_app/tree/main/lib/course/bloc

CourseApp: CourseEvent

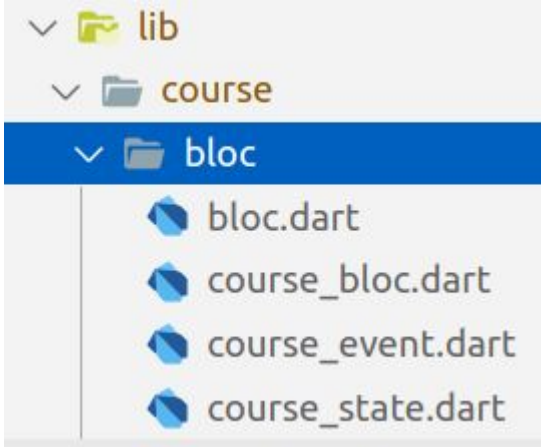
```
abstract class CourseEvent extends Equatable {
  const CourseEvent();
}

class CourseLoad extends CourseEvent {…

class CourseCreate extends CourseEvent {…

class CourseUpdate extends CourseEvent {…

class CourseDelete extends CourseEvent {…
```



Four **`Events`** corresponding to the four **`CRUD`** operations

CourseApp: CourseState

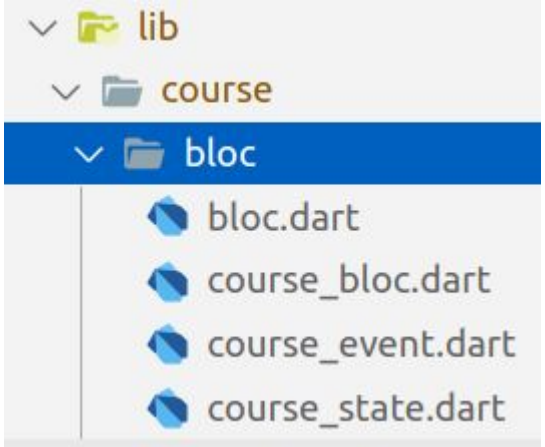
```
class CourseState extends Equatable {…

class CourseLoading extends CourseState {}

class CoursesLoadSuccess extends CourseState {
  final List<Course> courses;

  CoursesLoadSuccess([this.courses = const []]);

  @override
  List<Object> get props => [courses];
}

class CourseOperationFailure extends CourseState {}
```



Three **States**

CourseApp: CourseBloc

```
class CourseBloc extends Bloc<CourseEvent, CourseState> {
  final CourseRepository courseRepository;

  CourseBloc({@required this.courseRepository}) :
                                  super(CourseLoading());

  @override
  Stream<CourseState> mapEventToState(CourseEvent event) async* {
    if (event is CourseLoad) {…
    }
    if (event is CourseCreate) {…
    }
    if (event is CourseUpdate) {…
    }
    if (event is CourseDelete) {…
    }
  }
}
```



When the CRUD events happen, it generates `CourseLoadSuccess` state with the new course list object

Barrel Files

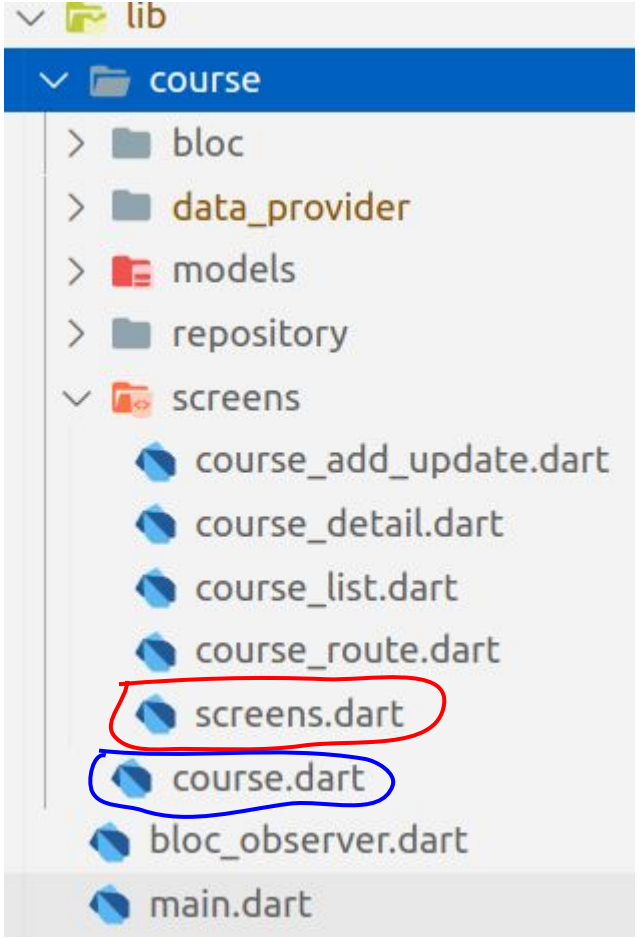
You can manage imports using **`export`**

In each directory there is a `.dart` file named after the directory name such as

`screens.dart` and `course.dart`

course > screens > screens.dart

```
export './course_list.dart';
export './course_add_update.dart';
export './course_detail.dart';
export './course_route.dart';
```



Barrel Files

These barrel files allows you to manage your imports nicely

You can minimize the number of imports in your files

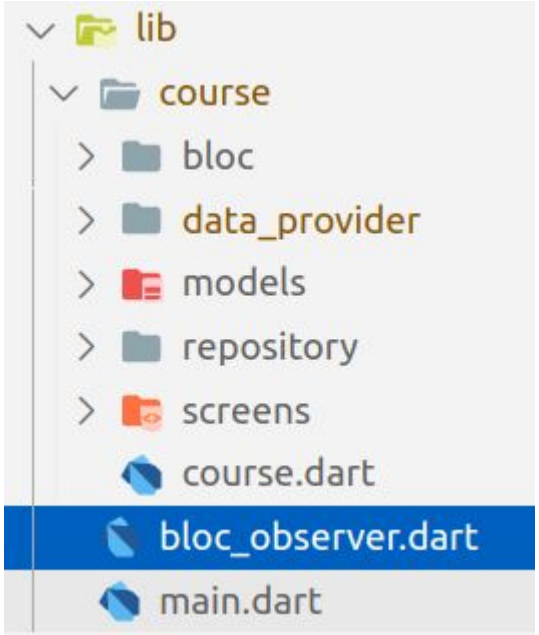
main.dart > ...

```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:flutter_network/bloc_observer.dart';
import 'package:flutter_network/course/course.dart';
import 'package:http/http.dart' as http;

Run | Debug
void main() {
  Bloc.observer = SimpleBlocObserver();
```

BlocObserver

bloc_observer.dart

Contains a class named **SimpleBlocObserver** which allows you to observe the transactions, for example for debugging purpose




```
I/flutter ( 2594): onEvent CourseLoad
I/flutter ( 2594): onTransition Transition { currentState: CourseLoading, event: CourseLoad, nextState: CourseLoading }
I/flutter ( 2594): onTransition Transition { currentState: CourseLoading, event: CourseLoad, nextState: CoursesLoadSuccess }
```

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/bloc_observer.dart

CourseApp

It is the root widget

It takes the **CoureRepostory** object as constructor argument

```
class CourseApp extends StatelessWidget {
  final CourseRepository courseRepository;

  CourseApp({@required this.courseRepository})
      : assert(courseRepository != null);

  @override
  Widget build(BuildContext context) {…
}
```

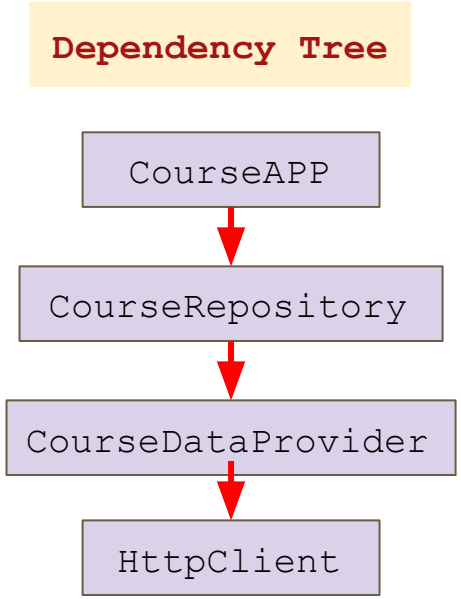
main.dart

The code on **line 8**, registers the **`BlocObserver`**

```
7   void main() {
8     Bloc.observer = SimpleBlocObserver();
9
10    final CourseRepository courseRepository = CourseRepository(
11      dataProvider: CourseDataProvider(
12        httpClient: http.Client(),
13      ), // CourseDataProvider
14    ); // CourseRepository
15
16    runApp(
17      CourseApp(courseRepository: courseRepository),
18    );
19  }
```

main.dart

The code on **line 10 to 14** injects the data provider to the repository and the http client to the data provider, then creates **`CourseRepository`** instance




```
7   void main() {
8     Bloc.observer = SimpleBlocObserver();
9
10    final CourseRepository courseRepository = CourseRepository(
11      dataProvider: CourseDataProvider(
12        httpClient: http.Client(),
13      ), // CourseDataProvider
14    ); // CourseRepository
15
16    runApp(
17      CourseApp(courseRepository: courseRepository),
18    );
19  }
```

main.dart

The code on **line 17** injects the course repository to the **`CourseApp`** and creates **`CourseApp`** instance

```
7   void main() {
8     Bloc.observer = SimpleBlocObserver();
9
10    final CourseRepository courseRepository = CourseRepository(
11      dataProvider: CourseDataProvider(
12        httpClient: http.Client(),
13      ), // CourseDataProvider
14    ); // CourseRepository
15
16    runApp(
17      CourseApp(courseRepository: courseRepository),
18    );
19  }
```

Providing the Repository to the widget tree

The build method of `CourseApp` widget returns a `MaterialApp` widget

The `CourseRepository` instance is provided above the `MaterialApp`

```
class CourseApp extends StatelessWidget {
  final CourseRepository courseRepository;

  CourseApp({@required this.courseRepository})
      : assert(courseRepository != null);

  @override
  Widget build(BuildContext context) {
    return RepositoryProvider.value(
      value: this.courseRepository,
```

Providing the Bloc to the widget tree

The `CourseBloc` instance is also provided above the `MaterialApp`

```
@override
Widget build(BuildContext context) {
  return RepositoryProvider.value(
    value: this.courseRepository,
    child: BlocProvider(
      create: (context) => CourseBloc(courseRepository: this.courseRepository)
        ..add(CourseLoad()),
      child: MaterialApp(
```

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/main.dart

Generating the `CourseLoad` Event

Inside the `CourseApp` while providing the `CourseBloc` to the tree we can generate the first `CourseLoad` event

```
@override
Widget build(BuildContext context) {
  return RepositoryProvider.value(
    value: this.courseRepository,
    child: BlocProvider(
      create: (context) => CourseBloc(courseRepository: this.courseRepository)
        ..add(CourseLoad()),
      child: MaterialApp(
```

Consuming `CourseLoadSuccess` state

The default route for the application is **`CourseList`** screen

Inside this widget we check if the generated state is `CourseLoadSuccess` and if it is, we display a list of course items inside a `ListView` as shown in the next slide

When each item is tapped it navigates to the `CourseDetail` screen by passing the current item as route argument

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_list.dart

```
body: BlocBuilder<CourseBloc, CourseState>(
  builder: (_, state) {
    if (state is CourseOperationFailure) {
      return Text('Could not do course operation');
    }

    if (state is CoursesLoadSuccess) {
      final courses = state.courses;

      return ListView.builder(
        itemCount: courses.length,
        itemBuilder: (_, idx) => ListTile(
          title: Text('${courses[idx].title}'),
          subtitle: Text('${courses[idx].code}'),
          onTap: () => Navigator.of(context)
              .pushNamed(CourseDetail.routeName, arguments: courses[idx]),
        ), // ListTile
      ); // ListView.builder
```

UpdateCourse/CreateCourse Events

In the AddUpdateCourse widget you can find how the UpdateCourse or CreateCourse events are generated when the save button is pressed

https://github.com/betsegawlemma/course_flutter_app/blob/main/lib/course/screens/course_add_update.dart

DeleteCourse Event

Inside the `CourseDetail` widget when the delete button is pressed `CourseDelete` event is generated

```
IconButton(
    icon: Icon(Icons.delete),
    onPressed: () {
      context.read<CourseBloc>().add(CourseDelete(this.course));
      Navigator.of(context).pushNamedAndRemoveUntil(
          CoursesList.routeName, (route) => false);
    }), // IconButton
```

References

https://flutter.dev/docs/cookbook (Networking and Forms section)

https://bloclibrary.dev/