Module 4

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 4 Study Guide and Deliverables

Topics:	• Local Databases Using Room • Shared Preferences • File Stporage • Content Providers
Readings:	• Online lectures • Notes and references cited in lectures • Chapters 15-16 from the "Head First" textbook (or corresponding chapters related to topics covered in this module in other textbooks)
Discussions:	• Discussion 4 ◦ Initial post due **Tuesday, August 2 at 6:00am ET** ◦ Respond to threads posted by others due **Thursday, August 4 at 6:00am ET**
Labs:	• Lab 4 due **Tuesday, August 2 at 6:00am ET**
Assignments:	• Assignment 4 due **Tuesday, August 2 at 6:00am ET**
Assessments:	• Quiz 4 due **Tuesday, August 2 at 6:00am ET**
Live Classrooms:	• **Monday, August 1 from 6:00-8:00pm ET**

Learning Outcomes

By the end of this module, you will be able to:

- Describe what is a SQLiteDatabase and basic usage.
- Describe Room Database and how it is used.
- Describe SharedPrreferences and how it is used.
- Describe different file directories and their usage.
- Describe what is a content provider and how to access a content provider.
- Correctly implement Android applications that can store the data in a database and sharedpreference.

Introduction

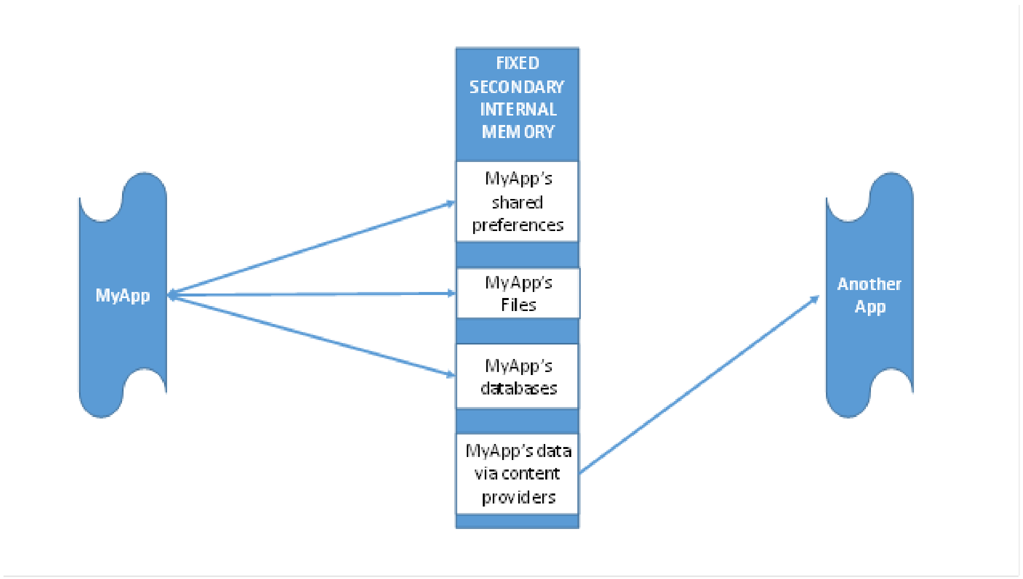
Every application needs to process and store some data. Mostly this will involve some persistent data stored in permanent storage units (e.g. internal device flash memory, external SD card, or cloud storage). In the MVC (Model View Controller) or MVVM (Model View ViewModel) architecture, the model subsystem defines all application persistent data and business logic. Model classes are usually first defined in POJO (Plain Old Java Objects) and then mapped to the database schema. Besides structured data in a database, there are also other means to store data. For example, persistent application settings or user preferences across user sessions need to be stored and remembered when the application is relaunched. Additional structured application data or unstructured data are stored in files (e.g. as an email attachment). In general, there are three types of data storage for each application:

- SQlite databases are used to store main application structured data.
- Shared preferences are used to store application settings, user session data, and other relatively small amounts of data.
- Additional data may be stored as files, either in the internal data partition, or external SD card partition.

To provide isolation and security, each application runs in its own sandbox. They can not access each other's data by default. Each application stores its data in a separate directory under /data/data, which is inaccessible to other applications. Each application data folder is named with its full package name. Each application data folder (/data/data/<appfullpackagename>) usually has the following subfolders: shared_pref, lib, files, and databases.

If you need to share data with some other applications, you can use the ContentProvider interface to share it explicitly with others.

The following figure summarizes this:


FIXED SECONDARY INTERNAL MEMORY
MyApp's shared preferences
MyApp's Files
MyApp's databases
MyApp's data via content providers
MyApp
Another App

In this module, we will answer the following questions for each type of data and storage:

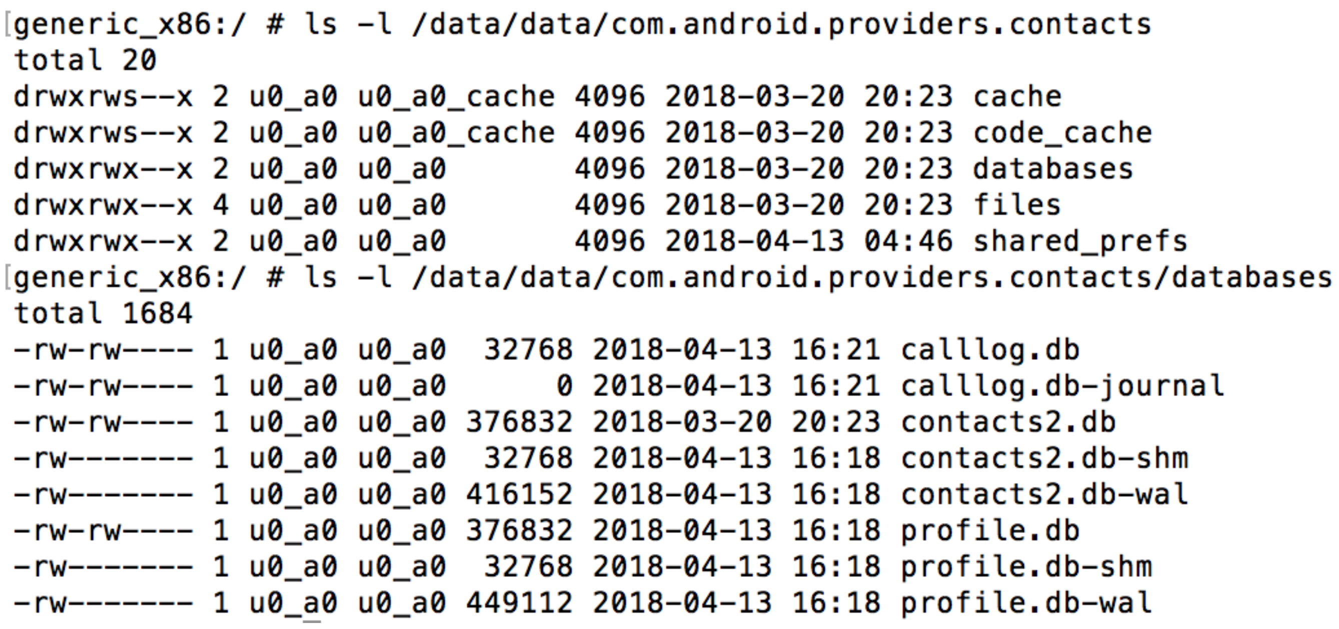
- What data should be stored in each type of storage?
- What formats are used to store these data? Where are these data stored?
- What APIs are used to read and write these data?
- What are common pitfalls (and security problems) when using these APIs?
- What are limitations?

Topic 1: Local Databases Using Room

Local Databases Using Room

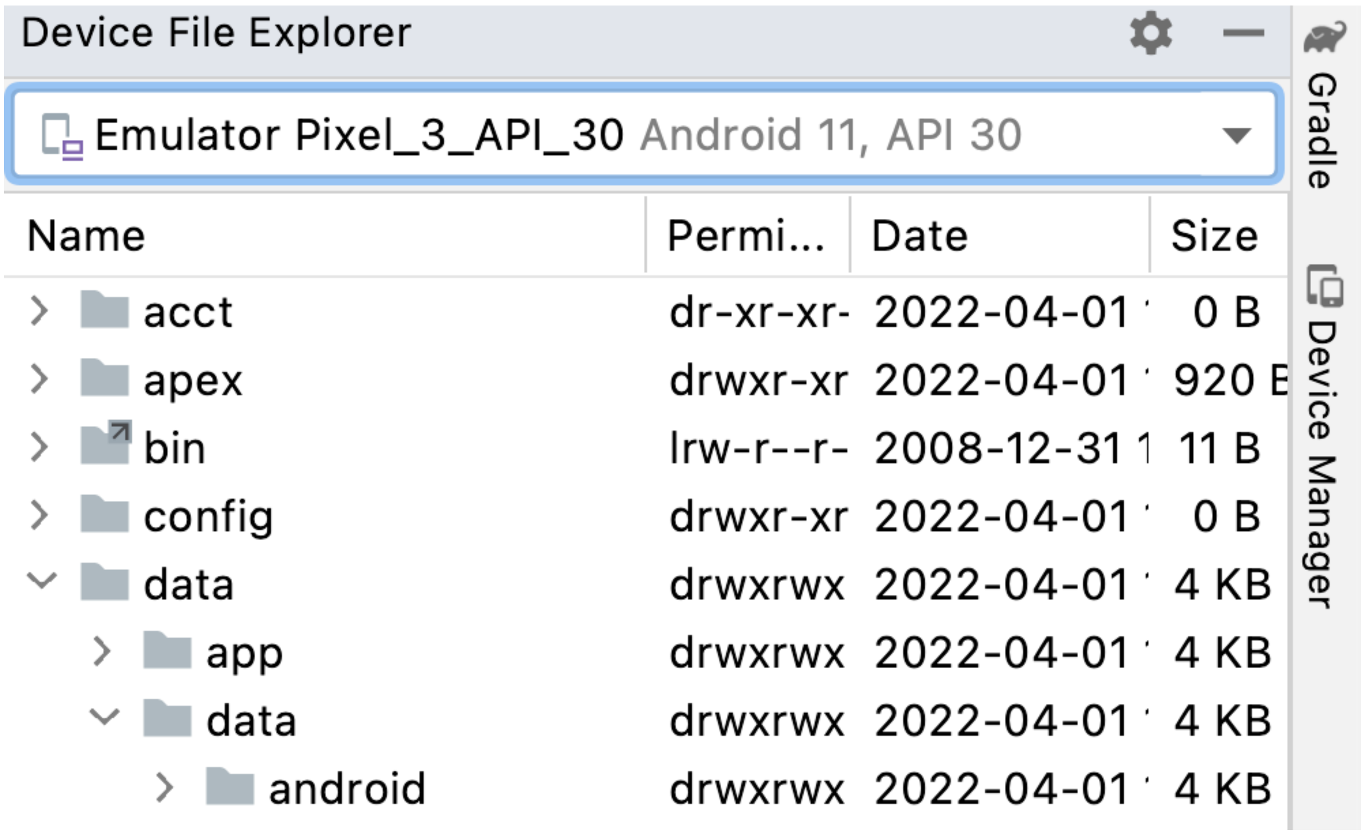
Most application data is stored on the device as structured data in the database. Android uses the SQLite Database as its default database. SQLite is an open source, in-process library that implements a self-contained, zero configuration and transactional SQL database engine. It is a very portable and lightweight database engine. It does not have the client-server relationship, and there is no need to install it. It is simply a single, cross-platform file containing multiple tables, triggers and views. It is very popular on many mobile platforms. As a condensed version of SQL (Structured Query Language), SQLite supports the standard SQL syntax and database transactions.

As mentioned before, each application stores its data in its own application folder, /data/data/<appfullpackagename>. This folder is private to the application and not accessible by others. Under this folder, there are usually several subfolders. SQLite database files are stored in the **`./databases`** subfolder. For example, the call log and contact information are stored in the database of the system (pre-installed) app com.android.providers.contacts:



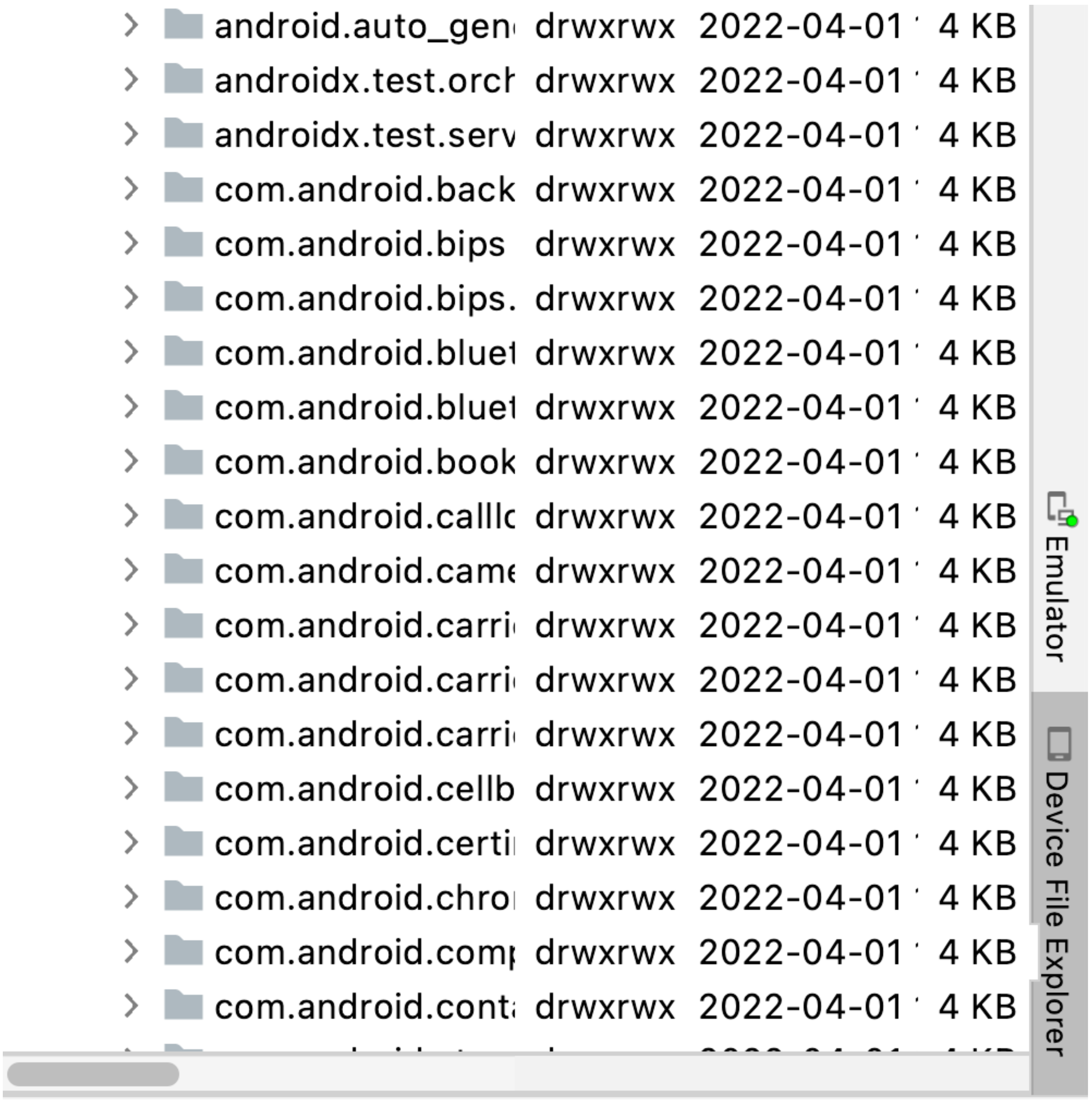
```
generic_x86:/ # ls -l /data/data/com.android.providers.contacts
total 20
drwxrws--x 2 u0_a0 u0_a0_cache 4096 2018-03-20 20:23 cache
drwxrws--x 2 u0_a0 u0_a0_cache 4096 2018-03-20 20:23 code_cache
drwxrwx--x 2 u0_a0 u0_a0       4096 2018-03-20 20:23 databases
drwxrwx--x 4 u0_a0 u0_a0       4096 2018-03-20 20:23 files
drwxrwx--x 2 u0_a0 u0_a0       4096 2018-04-13 04:46 shared_prefs
generic_x86:/ # ls -l /data/data/com.android.providers.contacts/databases
total 1684
-rw-rw---- 1 u0_a0 u0_a0  32768 2018-04-13 16:21 calllog.db
-rw-rw---- 1 u0_a0 u0_a0      0 2018-04-13 16:21 calllog.db-journal
-rw-rw---- 1 u0_a0 u0_a0 376832 2018-03-20 20:23 contacts2.db
-rw------- 1 u0_a0 u0_a0  32768 2018-04-13 16:18 contacts2.db-shm
-rw------- 1 u0_a0 u0_a0 416152 2018-04-13 16:18 contacts2.db-wal
-rw-rw---- 1 u0_a0 u0_a0 376832 2018-04-13 16:18 profile.db
-rw------- 1 u0_a0 u0_a0  32768 2018-04-13 16:18 profile.db-shm
-rw------- 1 u0_a0 u0_a0 449112 2018-04-13 16:18 profile.db-wal
```

We can use **Device File Explorer** in Android studio to view the application data folder of the one you are developing by clicking the Device File Explorer button in the tool window bar to open it. For example, we can view the Projectportal data folder at /data/data/edu.bu.projectportal.

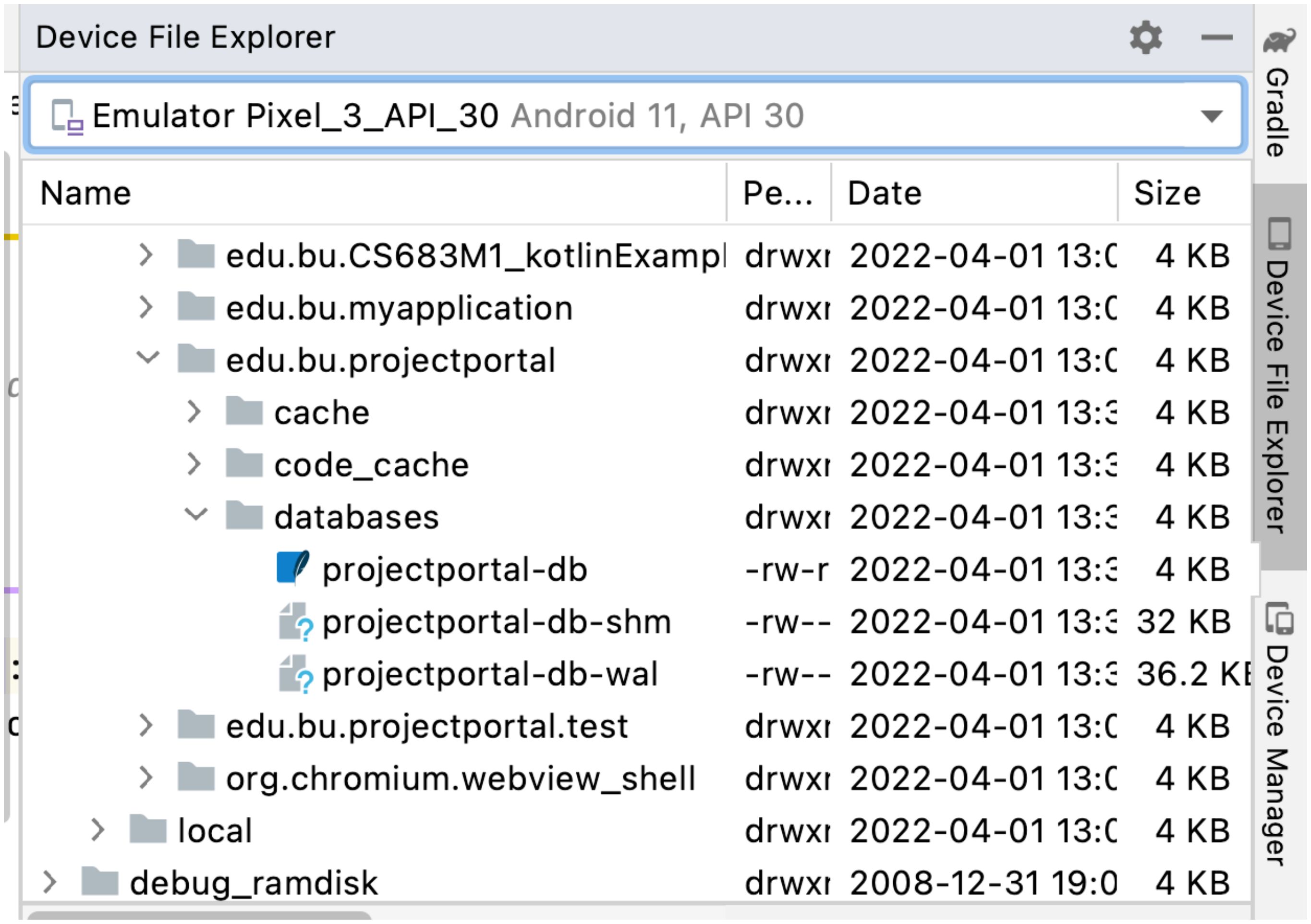


Emulator Pixel_3_API_30 Android 11, API 30

Name	Permi...	Date	Size
acct	dr-xr-xr-	2022-04-01	0 B
apex	drwxr-xr	2022-04-01	920 B
bin	lrw-r--r-	2008-12-31 1	11 B
config	drwxr-xr	2022-04-01	0 B
data	drwxrwx	2022-04-01	4 KB
app	drwxrwx	2022-04-01	4 KB
data	drwxrwx	2022-04-01	4 KB
android	drwxrwx	2022-04-01	4 KB



```
android.auto_gen  drwxrwx  2022-04-01  4 KB
androidx.test.orch  drwxrwx  2022-04-01  4 KB
androidx.test.serv  drwxrwx  2022-04-01  4 KB
com.android.back  drwxrwx  2022-04-01  4 KB
com.android.bips  drwxrwx  2022-04-01  4 KB
com.android.bips.  drwxrwx  2022-04-01  4 KB
com.android.bluet  drwxrwx  2022-04-01  4 KB
com.android.bluet  drwxrwx  2022-04-01  4 KB
com.android.book  drwxrwx  2022-04-01  4 KB
com.android.calllc  drwxrwx  2022-04-01  4 KB
com.android.came  drwxrwx  2022-04-01  4 KB
com.android.carri  drwxrwx  2022-04-01  4 KB
com.android.carri  drwxrwx  2022-04-01  4 KB
com.android.carri  drwxrwx  2022-04-01  4 KB
com.android.cellb  drwxrwx  2022-04-01  4 KB
com.android.certi  drwxrwx  2022-04-01  4 KB
com.android.chro  drwxrwx  2022-04-01  4 KB
com.android.comp  drwxrwx  2022-04-01  4 KB
com.android.conta  drwxrwx  2022-04-01  4 KB
```



Name	Pe...	Date	Size
edu.bu.CS683M1_kotlinExampl	drwxr	2022-04-01 13:0	4 KB
edu.bu.myapplication	drwxr	2022-04-01 13:0	4 KB
edu.bu.projectportal	drwxr	2022-04-01 13:0	4 KB
cache	drwxr	2022-04-01 13:3	4 KB
code_cache	drwxr	2022-04-01 13:3	4 KB
databases	drwxr	2022-04-01 13:3	4 KB
projectportal-db	-rw-r	2022-04-01 13:3	4 KB
projectportal-db-shm	-rw--	2022-04-01 13:3	32 KB
projectportal-db-wal	-rw--	2022-04-01 13:3	36.2 K
edu.bu.projectportal.test	drwxr	2022-04-01 13:0	4 KB
org.chromium.webview_shell	drwxr	2022-04-01 13:0	4 KB
local	drwxr	2022-04-01 13:0	4 KB
debug_ramdisk	drwxr	2008-12-31 19:0	4 KB

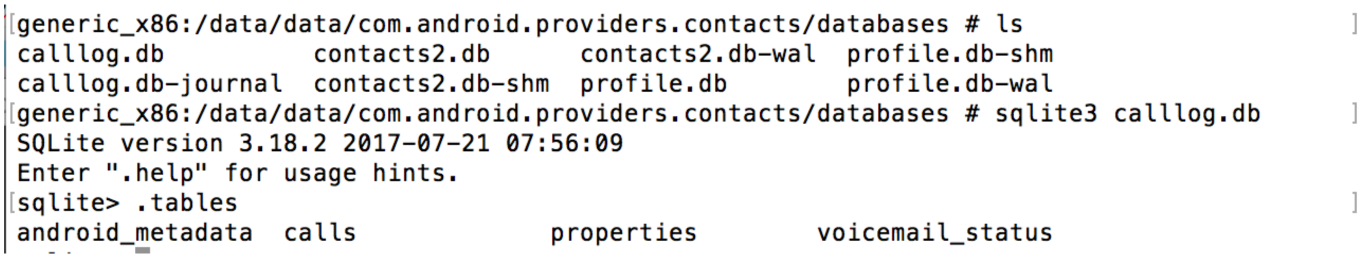
SQLite Basics

SQLite Database

Files Here, each .db file defines a separate database. The SQLite database files often use .db as extension names. However, they can also use other extension names such as .sqlite, .sqlitedb, or even no extension name.

A single SQLite database file defines a database that organizes the data into tables. The row of a table represents a single record and the column of the table represents a single field. A field in a SQLite database can be in one of the following data types: **`NULL, INTEGER, REAL, TEXT,`** and **`BLOB`**. **`BLOB`** is used to store a large array of binary data (bytes).

To view the content of a SQLite database file, you can use the SQLite command-line utility directly, which is included in Android, Mac and Linux by default. For example, you can use the command `"sqlite3 calllog.db"` to load the call log database and use the `".tables"` command to show all tables in this database:



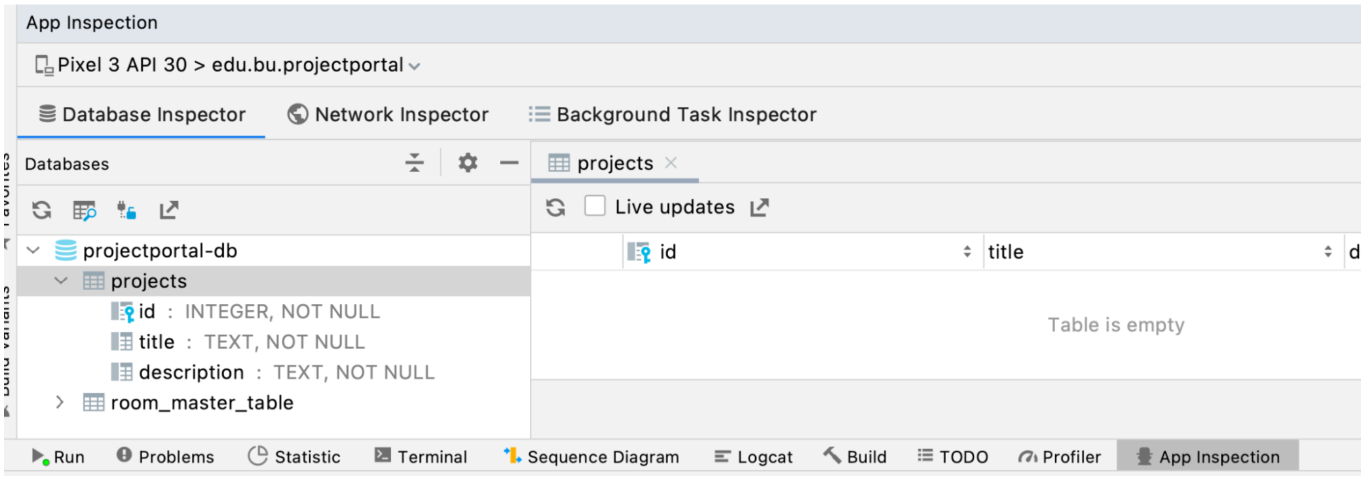
```
generic_x86:/data/data/com.android.providers.contacts/databases # ls
calllog.db          contacts2.db      contacts2.db-wal  profile.db-shm
calllog.db-journal  contacts2.db-shm  profile.db        profile.db-wal
generic_x86:/data/data/com.android.providers.contacts/databases # sqlite3 calllog.db
SQLite version 3.18.2 2017-07-21 07:56:09
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  calls             properties        voicemail_status
```

Here are some basic SQLite3 Commands:

- Connecting to a database: sqlite3 filename, e.g. `sqlite3 calllog.db`
- Dot commands:
  - .tables: shows all tables defined in the database
  - .schema table-name: shows schema of a particular table
  - .dump table-name: dumps the content of a particular table
  - .help: shows help information
  - .exit: disconnects from the database
- You can also execute SQL queries such as SELECT … FROM … WHERE … etc.

You can also install some GUI applications such as SQLite Browser to access the database.

In Android 4.0 or above, you can use **App inspector -> Database inspector** to check the database directly in Android studio. You can view table schemas, view and modify data in the tables, and run customized query using Database inspector.



projectportal-db
projects
id : INTEGER, NOT NULL
title : TEXT, NOT NULL
description : TEXT, NOT NULL
room_master_table

id	title	d

Table is empty

SQLite Database Operations

The Android SDK includes the SQLite software library that implements the SQL (Structured Query Language) database engine. The Android SDK also provides a number of APIs to define and manage your database in the application.

There are usually four basic functions for the persistent data, represented by the acronym CRUD. It stands for Create, Read, Update and Delete. The functions are mapped to an SQL statement in the relationship database as follows: C- INSERT, R- SELECT, U- UPDATE, D - DELETE

Here are some basic SQL statements:

- CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype,...);
- INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
- SELECT column1, column2, … FROM table_name WHERE condition;
- UPDATE table_name SET column1 = value1, column2 = value2, … WHERE condition;
- DELETE FROM table_name WHERE condition;
- DROP TABLE table_name;

While you can use raw query directly in Android applications, it is generally not a good idea. Instead you should use specific Android SQLite APIs for each CRUD operation.

Test Yourself

Where are databases of an application edu.bu.databaseexample stored on the device?

/data/data/edu.bu.databaseexample/databases/

Test Yourself

What are unique features of a SQLite database compared to other type of databases (such as MySQL)?

small, portable, no client-server, a single file containing everything

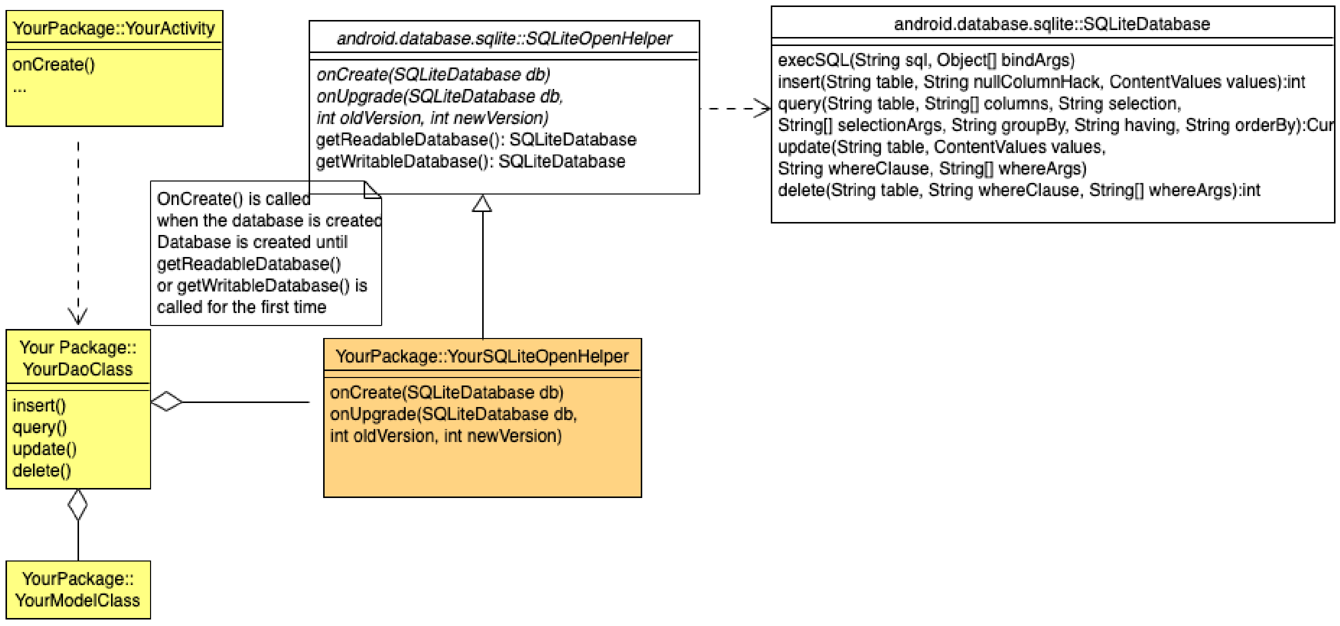
Test Yourself

What does CRUD stand for? List SQL statements for CRUD.

CRUD - (Create, Read, Update, Delete).

SQLite APIs

Android SQLite APIs have two important classes as shown in the following class diagram: `SQLiteOpenHelper` and `SQLiteDatabase`. `SQLiteDatabase` is a final class that cannot be subclassed. It provides a number of CRUD operations to operate databases. SQLiteOpenHelper is a class to help manage SQLiteDatabase. They are described below.


YourPackage::YourActivity
onCreate()
...
android.database.sqlite::SQLiteOpenHelper
onCreate(SQLiteDatabase db)
onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
getReadableDatabase(): SQLiteDatabase
getWritableDatabase(): SQLiteDatabase
android.database.sqlite::SQLiteDatabase
execSQL(String sql, Object[] bindArgs)
insert(String table, String nullColumnHack, ContentValues values):int
query(String table, String[] columns, String selection,
String[] selectionArgs, String groupBy, String having, String orderBy):Cur
update(String table, ContentValues values,
String whereClause, String[] whereArgs)
delete(String table, String whereClause, String[] whereArgs):int
OnCreate() is called when the database is created. Database is created until getReadableDatabase() or getWritableDatabase() is called for the first time
Your Package:: YourDaoClass
insert()
query()
update()
delete()
YourPackage::YourSQLiteOpenHelper
onCreate(SQLiteDatabase db)
onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
YourPackage:: YourModelClass

Class: SQLiteDatabase

Android defines a final class `SQLiteDatabase` to represent the SQLiteDatabase. A final class means that you **cannot** subclass this class. This class provides all CRUD operations that directly execute SQL statements using the underlying SQLite database engine. Since we as the application developers cannot modify the database engine, we cannot subclass this class to customize the database engine. Instead we can only write the wrapper methods to call these CRUD operations to perform on our database.

Each SQLite database in an application is associated with an SQLlite database (usually a .db) file stored under `/data/data/<appfullpackagename>/databases/`, as well as an SQLiteDatabase object in memory. We cannot directly create the SQLiteDatabase object. Instead, we need to call `getReadableDatabase()` or `getWritableDatabase()` from an SQLiteOpenHelper object. The database will be created if it did not exist before. In that case, the `onCreate()` method is called to create tables in the database by directly executing the SQL create statement. Specifically, we can call the `execSQL()` method of the database object to execute any raw SQL statements that have no returns, such as CREATE and DROP statements.

CRUD Operations

Android defines several methods in SQLiteDatabase to directly execute raw SQL statements. One is for the statements without return values such as CREATE and DROP statements: `execSQL(sql:string):unit`

Android also provides basic CRUD methods that execute specific CRUD statements as the following:

insert(table: String!, nullColumnHack: String!, values: ContentValues!): Long (insert data into the database)

query (table: String!, columns: Array<String!>!, selection: String!, selectionArgs: Array<String!>!, groupBy: String!, having: String!, orderBy: String!, limit:String!): Cursor (query data from the database based on some condition specified by the selection clause.)

update(table: String!, values: ContentValues!, whereClause: String!, whereArgs: Array<String!>!):Int (update data in the database based on some condition specified by the where clause.)

delete(table: String!, whereClause: String!, whereArgs: Array<String!>!) (delete data in the database based on some condition specified by the where clause.)

In these CRUD operations, two other classes are involved. `ContentValues` is a final class in Android to store a set of named values identified by keys. When we want to insert or update database tables, we need to specify both the column names and values. The ContentValues class helps us map the column value with the column name. It provides a set of put/get methods to add different types of values into the set or get values based on the key. Another important class is the `Cursor` interface. It provides random read-write access to the result set returned by a database query. When any query methods are called, a Cursor object is returned, which represents a number of records in the database.

Class: SQLiteOpenHelper

Android provides a class `SQLiteOpenHelper` to help class to manage database creation and version change. This class is an abstract class that defines several methods needed to open or create the database. We need to first create a subclass of SQLiteOpenHelper and implement two abstract methods `onCreate(SQLiteDatabase)` and `onUpgrade(SQLiteDatabase, int, int)`. You can also override some other methods such as `onDowngrade(SQLiteDatabase, int, int)` and `onOpen(SQLiteDatabase db)`.

call `getReadableDatabase()` or `getWritableDatabase()` to create and/or get the reference of a readable or writable database. If the database tables do not exist, this method will call `onCreate()` to create the database. After obtaining the reference to a database, we can perform CRUD operations on the database.

We will also need to define the database schema. This is usually defined in a contract class that defines table names, field names, and URI names as symbolic constants. This can be quite tedious. In addition, we usually need to create our own DAO (Database Access Object) class to encapsulate all related SQLiteDatabase CRUD methods.

Room Database

Instead of creating your own DAO class using SQLiteOpenHelper to SQLiteDatabase directly, there are a number of Android ORM (Object - Relational Mapping) packages that can help you handle your database and greatly simplify your database-related code. Here are a list of some well-designed ones:

- ROOM
- GreenDAO
- ActiveAndroid
- ORMLite
- OrmLite
- SugarORM
- DBFlow

Room Database APIs

Room Database provided by Google provides an abstraction layer over SQLite to make the database access much simpler and safer. It uses convenient annotations to minimize lots of boilerplate code, and provide compile time verification of SQL queries. It is highly recommended to use Room Database instead of SQLite APIs directly. Therefore, we omit all SQLite code examples and mainly focus on Room Database code.

To use Room database APIs, we need to add the following dependencies in the module build.gradle file

```
plugins {
   id 'com.android.application'
   id 'kotlin-android'
   id 'kotlin-kapt'
   id ("androidx.navigation.safeargs")
}

dependencies {
...

   def roomVersion = "2.4.2"

   implementation("androidx.room:room-runtime:$roomVersion")
   annotationProcessor("androidx.room:room-compiler:$roomVersion")
   // To use Kotlin annotation processing tool (kapt)
   kapt("androidx.room:room-compiler:$roomVersion")
 . . .
}
```

We will continue our ProjectPortal example with Room Database. Instead of using a static array “projects” to hold all project information, we will store all project information in the database table using the Room database. There are three components needed for the Room database.

- Data entities that represent tables in your app’s database. This is usually done by simply adding some annotations on your existing data class. For example, we can add “`@Entity`” annotation for the data class Project. You can also specify the table name. If not, the default table name is the same as the class name. Here we will change the table name to “projects”, instead of “Project”. By default all fields will be in the table. If you want to omit some fields, you can add “`@ignore`” annotation on a field. Each SQLite database table should have a primary key. If no primary key is specified, then the ROWID will be used as its primary key. To specify an explicit primary key, Here we use "`@PrimaryKey`" annotation to specify the id field to be the primary key. With the property autoGenerate as true, we can automatically generate the id. If the primary key is a combination of multiple fields, the annotation can be added by listing those columns in the `primaryKeys` property of @Entity. You can also specify a different column name for each field using @ColumnInfo(name = "something").After you add these annotations, Android studio will mostly show an error and provide you a quick fix to import necessary packages from and add the dependency.

```
@Entity (tableName = projects")
data class Project(
   @PrimaryKey(autoGenerate = true)
   val id: Int,
   var title: String,
   @ColumnInfo(name = )
   var description: String)
```

```
@Entity(primaryKeys = ["firstName", "lastName"])
data class User(
   val firstName: String,
   val lastName: String?
)
```

- Data access objects (DAOs) that provide methods that your app can use to query, update, insert, and delete data in the database. Usually we define a Dao as an interface, and must annotate it with `@Dao`. For example, the following `ProjectDao` provides methods >to perform CRUD operations on the projects table defined through the `@Entity` notation in the `Project` Data class. There are two types of DAO methods that define database interactions. One is the simple insert/update/delete methods without specifying SQL statements using simple annotations such as `@Insert`, `@Update`, or `@Delete`. The parameters should be a single or a collection of data entity class instances. The other is methods specified based on the SQL statements, particularly query methods. The parameters used in the methods are specified in the SQL queries using `:parameter`. For example, in the searchProject() method, there is one parameter `projId`, which is specified in the query as **`:projId`**. Depending on the query result, the methods usually return a collection of data entity class instances. Here we return **a livedata** of a list of projects, so that we can directly use it in the Viewmodel class, and in turn the UI controllers can be notified whenever data changes in the database.

```
@Dao
interface ProjectDao {
//     @Insert
//     fun addProjects(projects:List<Project>)

   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun addProject(project:Project)

   @Delete
   fun delProject(project:Project)

   @Update
   fun editProject(project: Project)

   @Query("SELECT count(*) From projects")
   fun count(): LiveData<Int>

   @Query("SELECT * FROM projects")
   fun getAllProjects(): LiveData<List<Project>>

   @Query("SELECT * FROM projects where id = :projId")
   fun searchProject(projId: Long): LiveData<Project>

   @Query("SELECT * FROM projects WHERE title like :projTitle ")
   fun searchProjectsbyTitle(projTitle:String): LiveData<List<Project>>
}
```

- The database class that holds the database and serves as the main access point for the underlying connection to your app's persisted data. We must define our database class as an abstract class that extends RoomDatabase and annotated with @Database annotation that includes an entities array that lists all of the data entities associated with the database. We also need to define an abstract method that has zero arguments and returns an instance of the DAO class for each DAO class that is associated with the database. For example the following code defines the ProjectPortalDatabase with a single entity Project and the related ProjectDao.

```
@Database(
   entities = [Project::class],
   version = 1
)
abstract class ProjectPortalDatabase: RoomDatabase() {
   abstract fun projectDao(): ProjectDao
}
```

As you can see here, all we need to define are interfaces which the other part of the app can use to access the database. The implementation of the Dao interfaces and the Database class will be automatically generated by the Room Database when we create an instance of the defined Database using the Room Database Builder. For example, the following code creates an instance of ProjectProtalDatabase with the database name as projectorportal-db. Then we can use the projectportalDatabase. projectDao() to get an instance of ProjectDao. In turn, we can call all the CRUD methods defined in ProjectDao() to access the projectportal-db database through the ProjectDao instance. For example, to add a project, we can call `projectportalDatabase.projectDao().addProject(project)`.

There are several places where we can create an instance of a room database. One is to create the room database as a single instance (as a companion object) in the Database class. Another is to create the database instance in the application class. We can use an application class to hold some global application data.

```kotlin
class ProjectPortalApplication: Application() {
   lateinit var projectportalDatabase: ProjectPortalDatabase
   override fun onCreate() {
       super.onCreate()

       projectportalDatabase =
           Room.databaseBuilder(
               applicationContext, ProjectPortalDatabase::class.java,
               "projectportal-db"
           ).build()
   }
}
```

There will be only one instance of this application class. Make sure to specify the application name as the defined application class name in the Manifest file, as shown below.

```xml
<application
   android:allowBackup="true"
   android:icon="@mipmap/ic_launcher"
   android:label="@string/app_name"
   android:roundIcon="@mipmap/ic_launcher_round"
   android:supportsRtl="true"
   android:name=".ProjectPortalApplication"
   android:theme="@style/Theme.ProjectPortal">
```

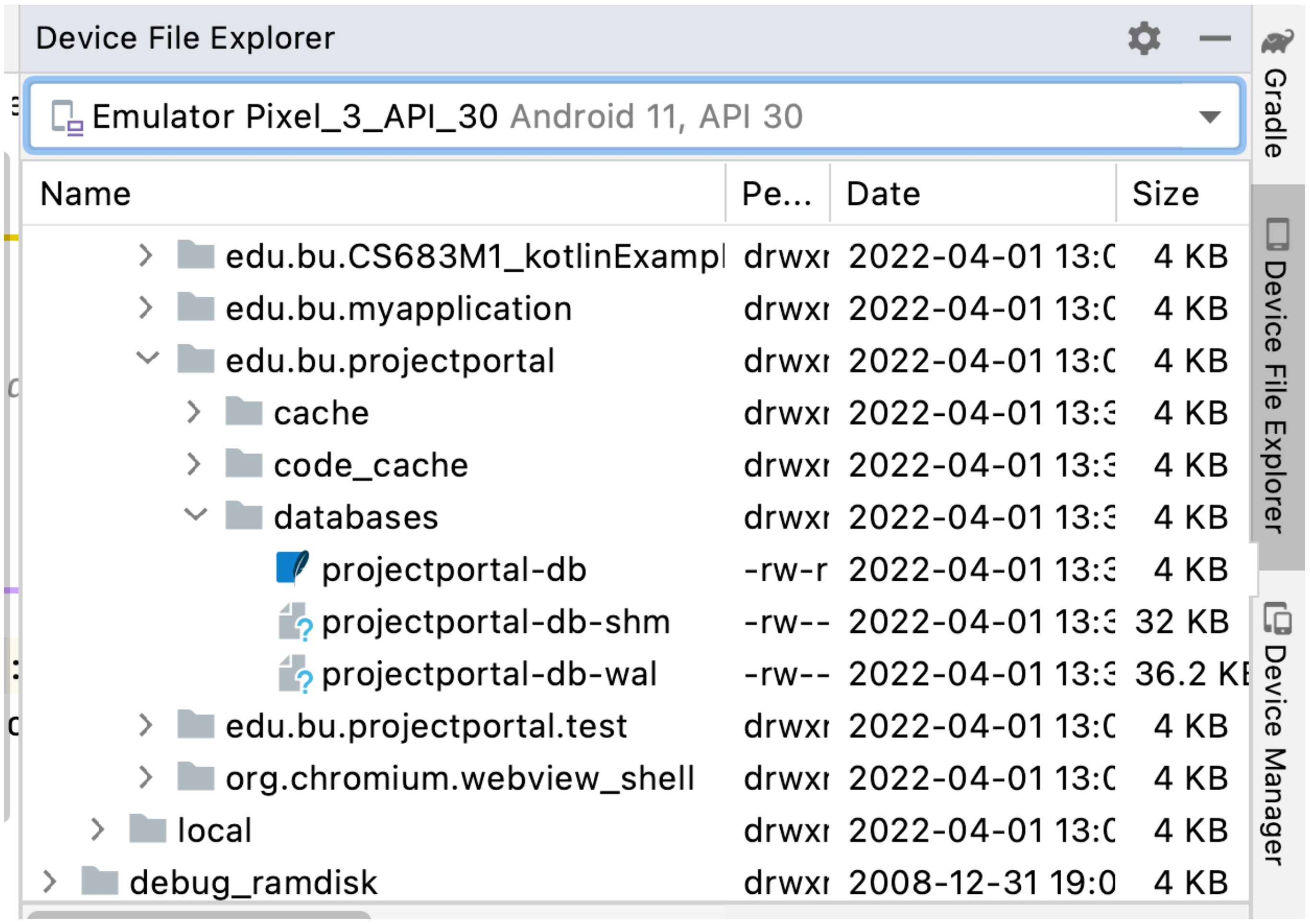
We can also add a callback function to add some initial data into the database for the testing purpose now. We will override the onCreate() method to add some data into the database when it is created. As shown below, we will add several projects into the database as the initial data.

```kotlin
class ProjectPortalApplication: Application() {
   lateinit var projectportalDatabase: ProjectPortalDatabase
   override fun onCreate() {
       super.onCreate()
       projectportalDatabase =
           Room.databaseBuilder(
               applicationContext, ProjectPortalDatabase::class.java,
               "projectportal-db"
           )
               // add a callback to modify onCreate() to
               // add some initial projects.
```

```
                .addCallback(object : RoomDatabase.Callback() {
                override fun onCreate(db: SupportSQLiteDatabase) {
                    super.onCreate(db)
                    addInitProjects()
                }
            })
                .build()
    }

    // this is only used to add some initial projects
    // when the database is first created.
    // this is done through
    fun addInitProjects(){
        // need to execute in a separate thread
        Executors.newSingleThreadScheduledExecutor().execute {
             projectportalDatabase.projectDao().addProject(
                Project(0,  "weather Forecast",
                "Weather Forecast is ...")
            )
             projectportalDatabase.projectDao().addProject(
                Project(0,  "Project Portal",
                 "Project Portal is ...")
            )
             projectportalDatabase.projectDao().addProject(
                Project(0,  "Connect Me",
                 "Connect me is ...")
            )
        }
    }
}
```

Please be aware that database operations such as add, delete, and update need to be *performed* ***asynchronously****, so here we* ***create a new executor*** to execute these insert operations. We will discuss asynchronous processing in the next Module.

When we run the app, we will see the projectportal-db file under /data/data/edu.projectportal/databases/projectportal-db.



Device File Explorer

Emulator Pixel_3_API_30 Android 11, API 30

Name	Pe...	Date	Size
edu.bu.CS683M1_kotlinExampl	drwxı	2022-04-01 13:(	4 KB
edu.bu.myapplication	drwxı	2022-04-01 13:(	4 KB
edu.bu.projectportal	drwxı	2022-04-01 13:(	4 KB
cache	drwxı	2022-04-01 13:3	4 KB
code_cache	drwxı	2022-04-01 13:3	4 KB
databases	drwxı	2022-04-01 13:3	4 KB
projectportal-db	-rw-r	2022-04-01 13:3	4 KB
projectportal-db-shm	-rw--	2022-04-01 13:3	32 KB
projectportal-db-wal	-rw--	2022-04-01 13:3	36.2 KI
edu.bu.projectportal.test	drwxı	2022-04-01 13:(	4 KB
org.chromium.webview_shell	drwxı	2022-04-01 13:(	4 KB
local	drwxı	2022-04-01 13:(	4 KB
debug_ramdisk	drwxı	2008-12-31 19:0	4 KB

The following diagram on the android developer website shows the relationship between these three components and the rest of the app.

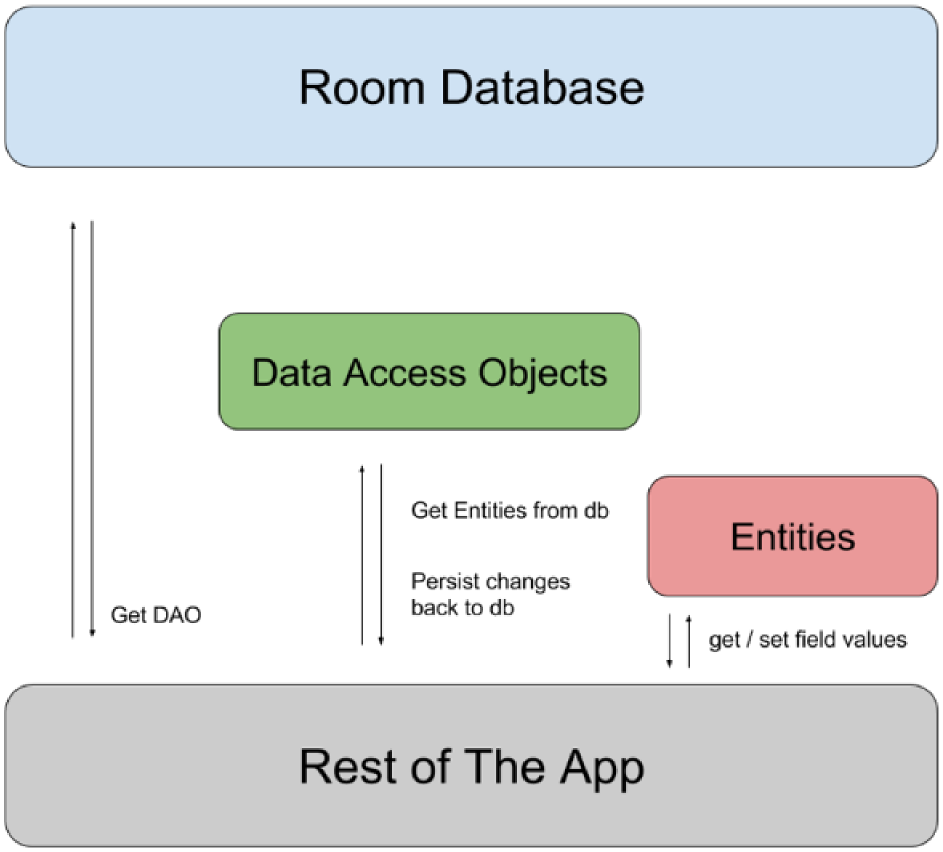


Figure 1. Diagram of Room library architecture. Accessing data using Room DAOs

Test Yourself

What are the three components of the Android Room database? What are the annotations for each of these components?

Data Entity class: @Entity, Dao Interface: @Dao, Database abstract Class: @Database

Test Yourself

How would you create a room database instance?

Use Room databaseBuider.

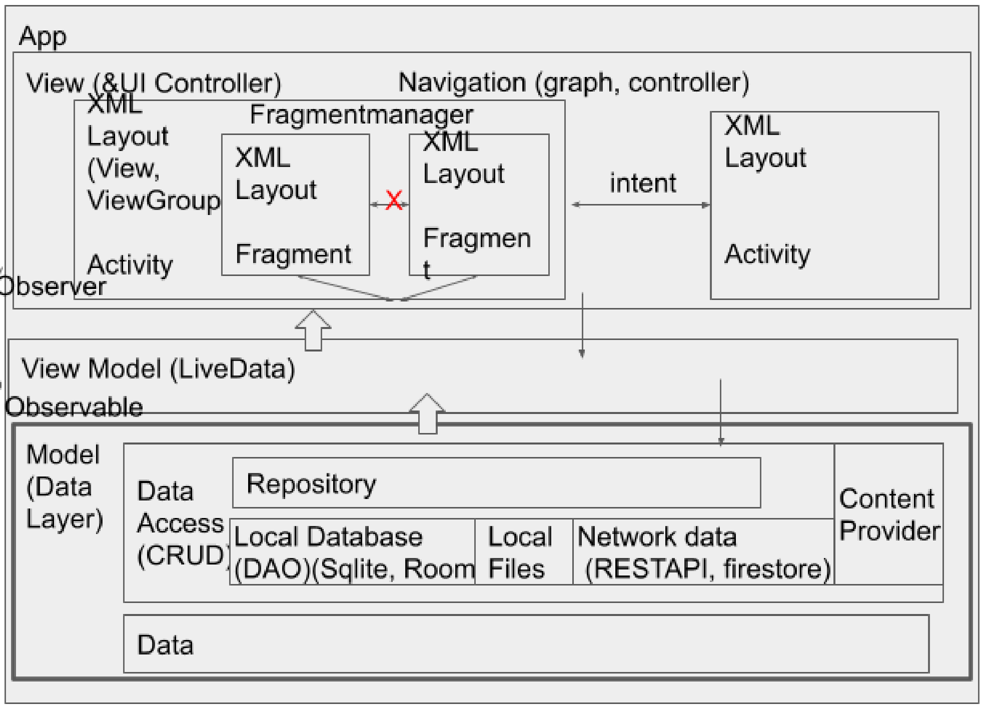
Test Yourself

Write down annotations for simple CRUD methods defined in the Dao interface.

(Please see notes.)

Android Application Architecture

In the last module, we have introduced MVVM architecture for Android applications. We discussed the role of ViewModels and LiveData to help manage UI data, so that UI controllers such as activities and fragments can be simpler. The VM layer will need to interact with the underneath data layer to access data in order to provide data to the UI layer. In this module, we will discuss the components in the data layer and how they interact with the upper layer. The following diagram shows all components in the architecture. In this module, we will focus on the data layer, particularly database and repository and how they interact with ViewModels.



Interact with ViewModel

We introduced ViewModel in the last Module. ViewModels are used to access the underneath data layer to provide data to the UI components. It is responsible to handle all UI data and it is lifecycle aware. So now, we will get data from the database instead of from a static variable.

Since we have stored the database instance in the application object, we can pass the application instance to the view model in order to access the database instance. We can use **AndroidViewModel** as the base class which has a parameter of an application object. The code below shows how to pass the application object and how to get the dao object through the passed application object. Here we will use the livedata returned from the database directly in this view model.

```
class ProjectListViewModel(application: Application):
AndroidViewModel(application) {
   // pass the projectportalApplication as a parameter
   // make sure to define the application name in the manifest file.
   val projectportalDatabase = (application as ProjectPortalApplication).projectportalData
   val projectDao = projectportalDatabase.projectDao()

   private val _projectList: LiveData<List<Project>> = projectDao.getAllProjects()

   val projectList:LiveData<List<Project>>
       get() = _projectList

   fun getAllProjects(): LiveData<List<Project>> {
       return projectDao.getAllProjects()
   }

   fun addProject(project: Project) {
       Executors.newSingleThreadExecutor().execute {
           projectDao.addProject(project)
       }
   }
}
```

Since we will use the data returned from the database as the data source for the adapter, we need to make sure the adapter data will be updated when the livedata returned from the database changes. Instead of passing the data source through the constructor method in the adapter, we will replace the data item whenever it changes.

```
class MyProjListRecyclerViewAdapter(
 //  private val projects: List<Project>,
   private val onProjectClickListener: OnProjectClickListener
)
   : RecyclerView.Adapter<MyProjListRecyclerViewAdapter.ViewHolder>() {
   private val projects = mutableListOf<Project>()
   //changes data source content
   fun replaceItems(myProjects: List<Project>) {
       projects.clear()
       projects.addAll(myProjects)
       notifyDataSetChanged()
   }
. . .
}
```

In the list fragment, we will not pass the data when creating the adapter, rather, we will update the adapter data source in the observer onChange() method, when observing the projectList live data in the ViewModel.

```
val myAdapter = MyProjListRecyclerViewAdapter(
           object : MyProjListRecyclerViewAdapter.OnProjectClickListener{
                   override fun onProjectClick(project: Project) {
                       viewModel.setCurProject(project)
                   }
               })

           this.adapter = myAdapter

           listViewModel.projectList.observe(viewLifecycleOwner, Observer{
               myAdapter.replaceItems(it)
               viewModel.initCurProject(myAdapter.getProject(0))

           })
```

Repository

Though local sqlite databases are most commonly used, we may use other methods to store structured data, e.g. using cloud data store or database on the server side. To minimize the changes needed in the other part of the application when changing the data storage, we can create a uniform interface to access data for the rest of the application no matter what data storage method is used. This can be achieved by adding repositories between data source and other parts of the application such as UI. The following diagram (Data layer) shows the data layer in a layered architecture for Android application.

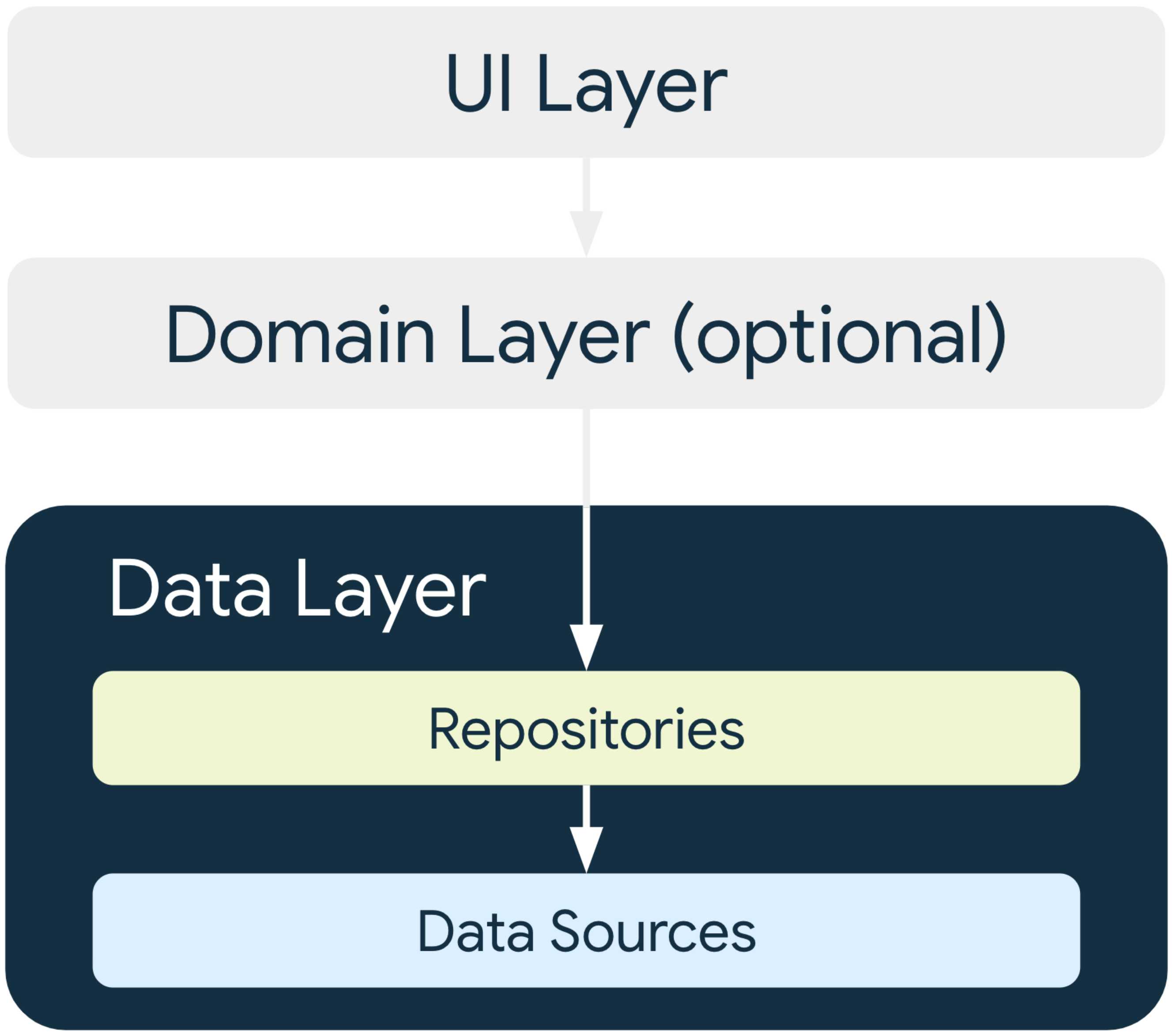


Figure 1: Data layer in app architecture.

Now, let us create a repository class to encapsulate all database operations, so that the UI layer (specifically viewmodels here) will not interact directly with the database, but the repository. If later, we want to change the data storage, we need to only change this class. The repository classes will be the entry points for the rest of the application to access the data. As shown in the code below, we can simply wrap the database operations in the repository.

```
package edu.bu.projectportal

import androidx.lifecycle.LiveData
import edu.bu.projectportal.datalayer.Project
import edu.bu.projectportal.datalayer.ProjectDao
import java.util.concurrent.Executors

class ProjectPortalRepository (
   private val projectDao: ProjectDao
) {
   val executor =  Executors.newSingleThreadExecutor()

   fun addProject(project: Project){
       executor.execute {
           projectDao.addProject(project)
       }
   }

   fun delProject(project: Project) {
       executor.execute {
           projectDao.delProject(project)
       }
   }

   fun editProject(project: Project) {
       executor.execute {
           projectDao.editProject(project)
       }
   }

   fun getAllProjects(): LiveData<List<Project>*gt; {
       return projectDao.getAllProjects()
   }

   fun searchProject(projId: Long): LiveData<Project> {
       return projectDao.searchProjectById(projId)
   }

   fun searchProjectsbyTitle(projTitle:String): LiveData<List<Project>> {
       return projectDao.searchProjectsByTitle(projTitle)
   }

   fun count(): LiveData<Int> {
       return projectDao.count()
   }
}
```

We can create a repository instance in the onCreate()method in the ProjectPortalApplication class as well.

```
projectPortalRepository =
   ProjectPortalRepository(projectportalDatabase.projectDao())
```

Now, the view models will interact with the repository object instead of the database directly.

```
class ProjectListViewModel(application: Application): AndroidViewModel(application) {
      val projectPortalRepository =
       (application as ProjectPortalApplication).projectPortalRepository

   fun getAllProjects(): LiveData<List<Project>> {
       return projectPortalRepository.getAllProjects()
   }
. . .
}
```

Create Additional UIs to Manipulate Data

Now, let us create some additional UI screens to perform CRUD operations. We will show the example of a project addition screen and a project deletion screen. You can try to work on the search screen by yourself.

Add a Project

First, we include an "add" button so that we can add projects. We will use a floating action button for this add button. We can add this button in the project list fragment, or on the main activity. Since currently we use the sliding pane, it is easier to add the button on the main activity. We will add in the left pane where the list fragment resides.

```
<?xml version="1.0" encoding="utf-8"?>

<androidx.slidingpanelayout.widget.SlidingPaneLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/slidepane"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <androidx.constraintlayout.widget.ConstraintLayout
       android:id="@+id/side_pane_content"
       android:layout_width="200dp"
       android:layout_height="match_parent"
       android:layout_gravity="start">
```

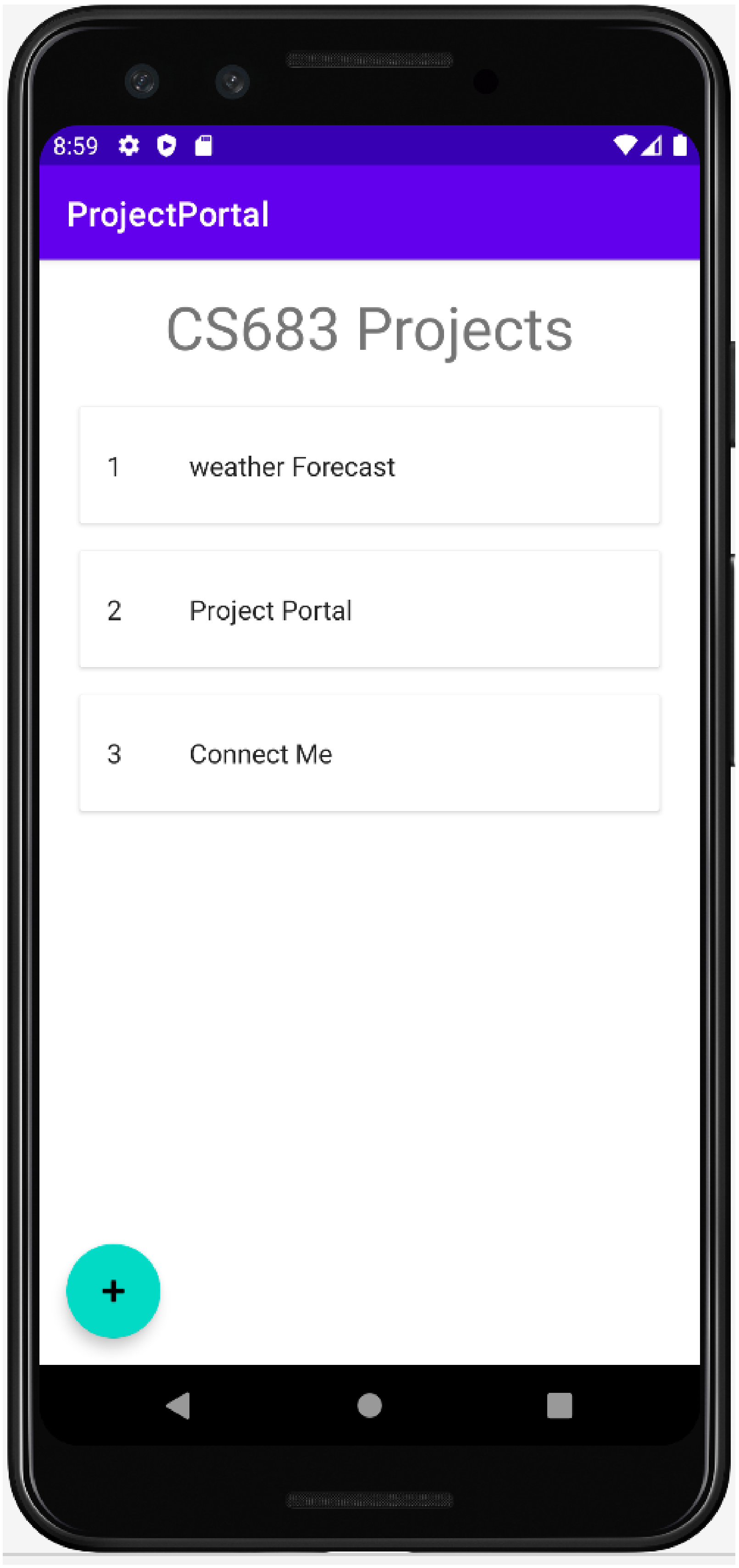
```
        <androidx.fragment.app.FragmentContainerView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:name="edu.bu.projectportal.fragments.ProjListRecyclerViewFragment"
            android:id="@+id/list_fragment" />

        <com.google.android.material.floatingactionbutton.FloatingActionButton
            android:id="@+id/fab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            android:layout_marginStart="16dp"
            android:layout_marginBottom="16dp"
            app:srcCompat="@android:drawable/ic_input_add" />

    </androidx.constraintlayout.widget.ConstraintLayout>

    <androidx.fragment.app.FragmentContainerView
        android:layout_width="300dp"
        android:layout_height="match_parent"
        android:id="@+id/detailContainerId"
        android:name="androidx.navigation.fragment.NavHostFragment"
        app:defaultNavHost="true"
        android:layout_weight="1"
        android:layout_gravity="end"
        app:navGraph="@navigation/nav_graph" />

</androidx.slidingpanelayout.widget.SlidingPaneLayout>
```

Then we implement the button onclick listener as below.

```
class MainActivity : AppCompatActivity(), MyProjListRecyclerViewAdapter.OnProjectClickList
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.content_slidepane)

findViewById<FloatingActionButton>(R.id.fab).setOnClickListener{

findViewById<FragmentContainerView>(R.id.detailContainerId)

?.findNavController()?.navigate(R.id.action_detailFragment_to_addFragment)
            findViewById<SlidingPaneLayout>(R.id.slidepane).open()
        }
    }
```

For the “add project” page, we can reuse the edit project page. We can simply reuse the xml layout file. We can even reuse the edit project fragment. But for simplicity, we will create another fragment for it. The code is actually pretty similar to the edit fragment. However, instead of trying to edit the current project, we need to add a project. You can find the detailed code in the example code zip file.

Here is the execution result on two devices with different screen sizes.



8:59

ProjectPortal

CS683 Projects

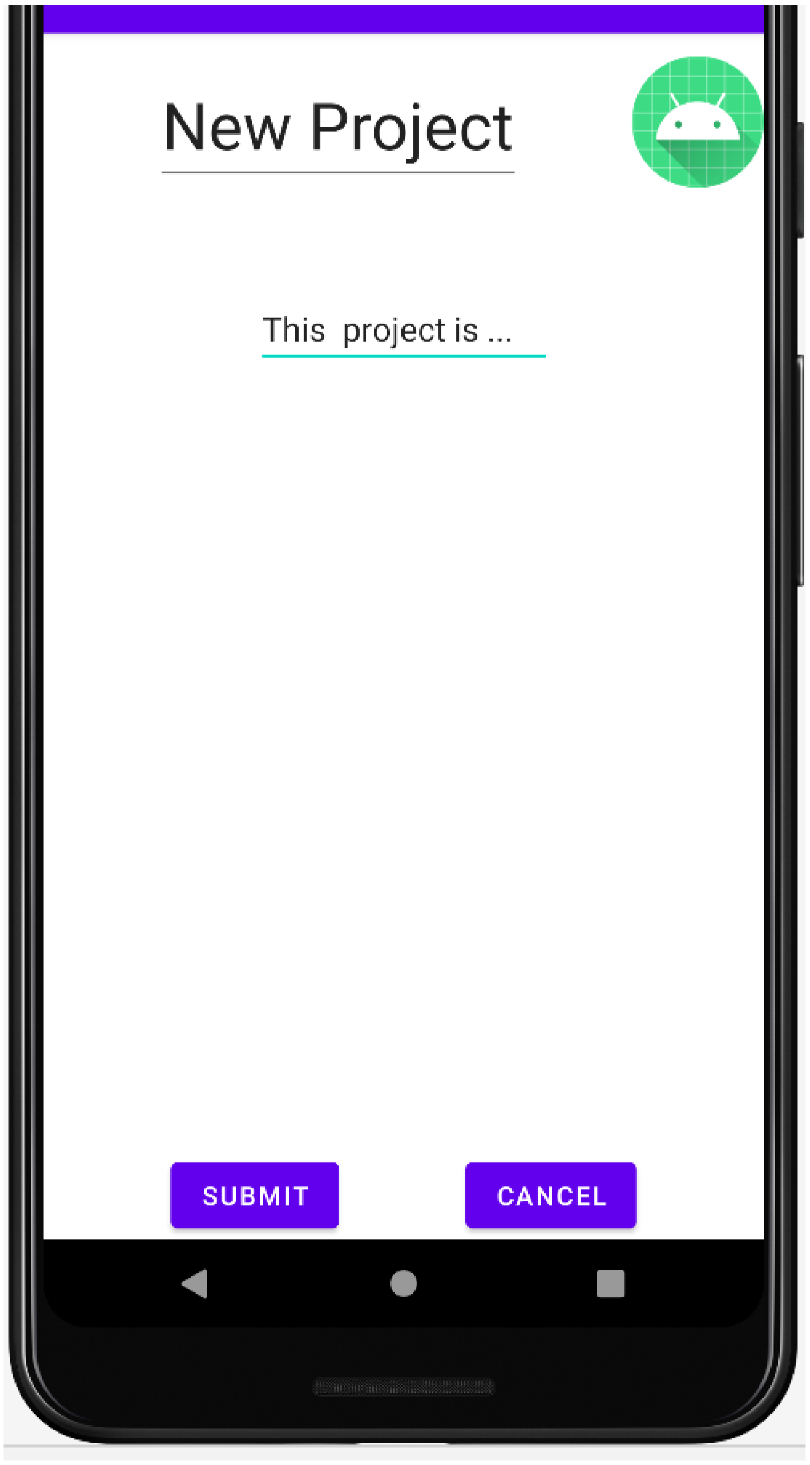
1 weather Forecast

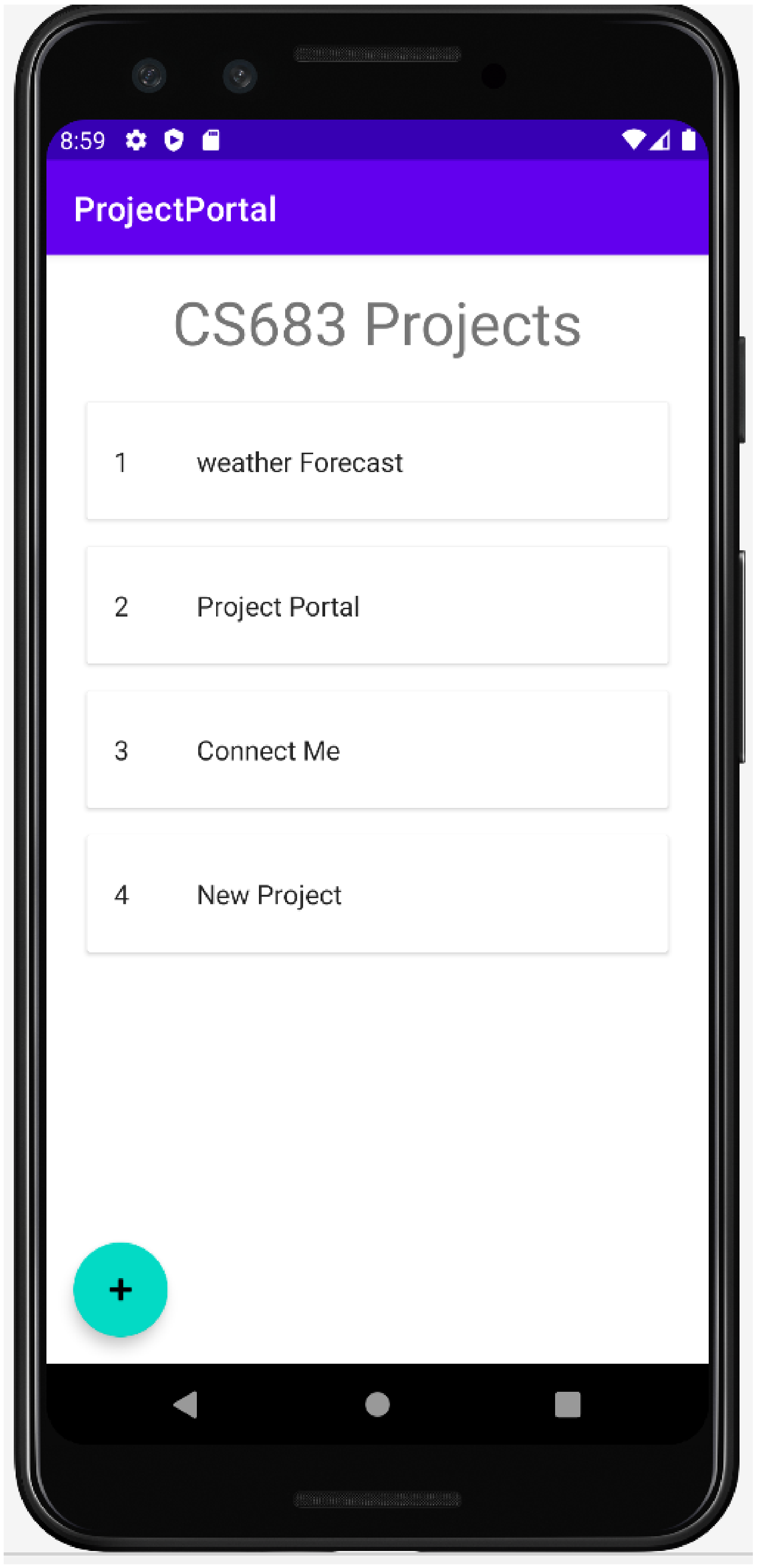
2 Project Portal

3 Connect Me

8:59

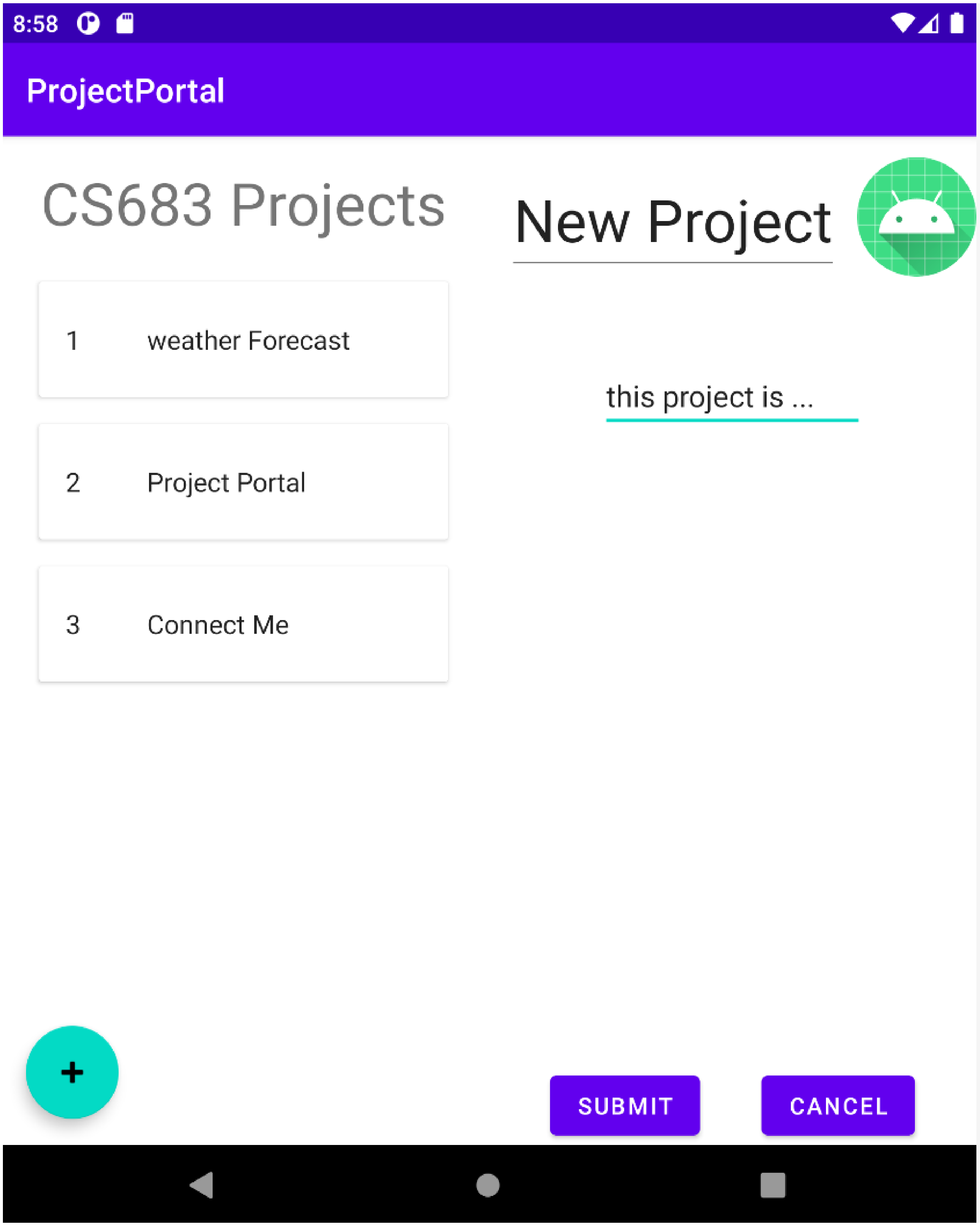
ProjectPortal


New Project
This project is ...
SUBMIT
CANCEL



CS683 Projects

1	weather Forecast
2	Project Portal
3	Connect Me
4	New Project



8:58

ProjectPortal

CS683 Projects

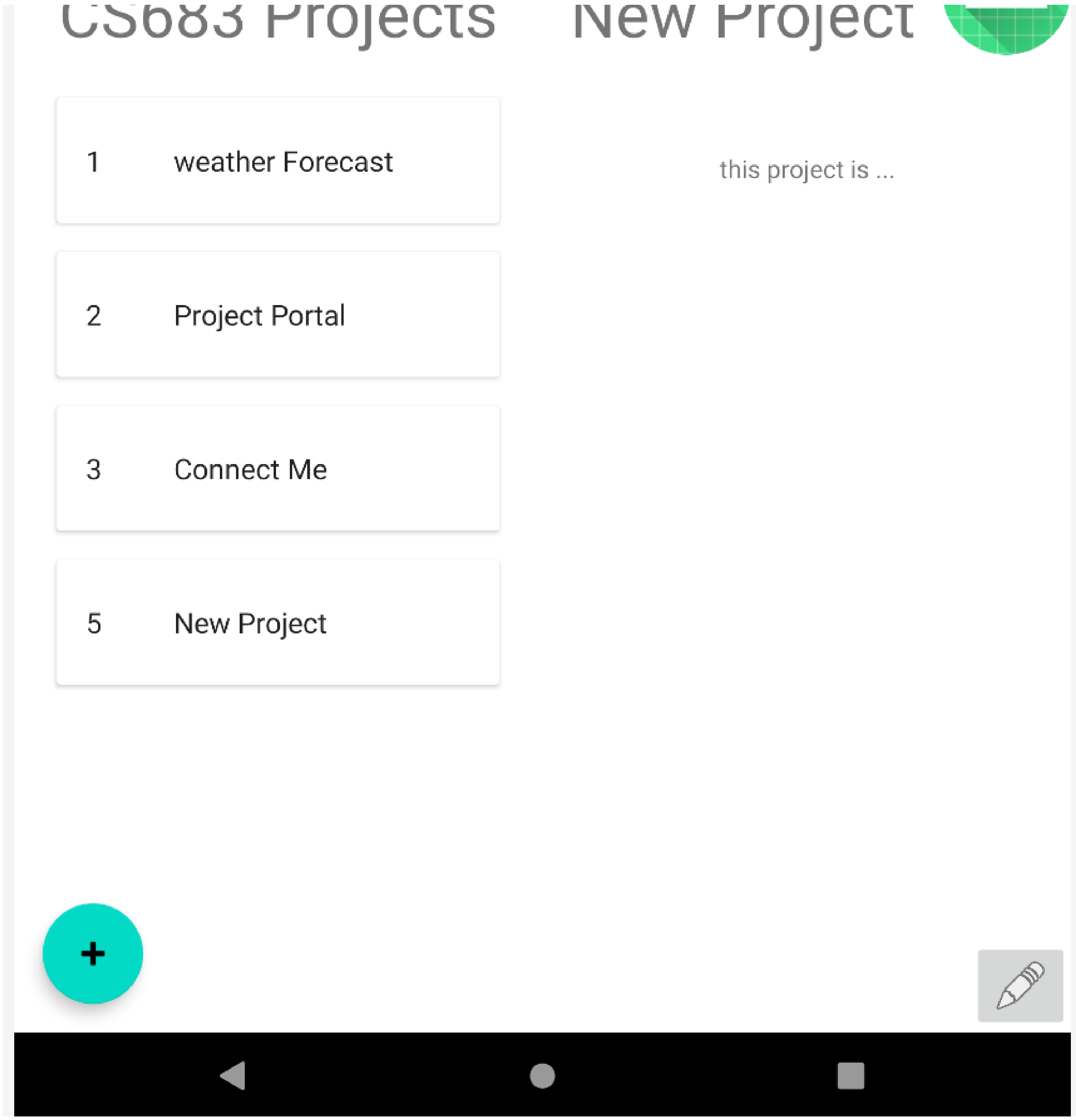
New Project

1	weather Forecast
2	Project Portal
3	Connect Me

this project is ...

8:58

ProjectPortal


CS683 Projects

New Project

1 weather Forecast

2 Project Portal

3 Connect Me

5 New Project

this project is ...

Delete Project

We want to delete unwanted projects. We can delete a project by sliding the project to the right in the recycler view. Here we use `ItemTouchHelper` defined in Recyclerview. It is a utility class to add swipe to dismiss and drag & drop support to RecyclerView. To create an ItemTouchHelper object, we need to provide a callback to specify how to control the behavior of this touch helper. We need to override three methods in the callback as shown below. The onSwapped() method implements the deletion action.

```
inner class SwipeToDeleteCallback: ItemTouchHelper.SimpleCallback(0 ,
    ItemTouchHelper.LEFT  or ItemTouchHelper.RIGHT ) {
    override fun  onMove (
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        target: RecyclerView.ViewHolder
    ): Boolean = false

    override fun  getMovementFlags (
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder
    ) = makeMovementFlags(
        ItemTouchHelper.ACTION_STATE_SWIPE ,
        ItemTouchHelper.RIGHT
    )

override fun  onSwiped (viewHolder: RecyclerView.ViewHolder, direction: Int) {
    val position = viewHolder.bindingAdapterPosition
    // get the project to be deleted
    val  project = myAdapter .getProject(position)
    // delete the project and update curProject livedata in the viewmodel
    // add your code here     }
}
```

Test Yourself

Complete the onSwiped() method in the previous code to delete the project.

(Please check the example code in the zip file.)

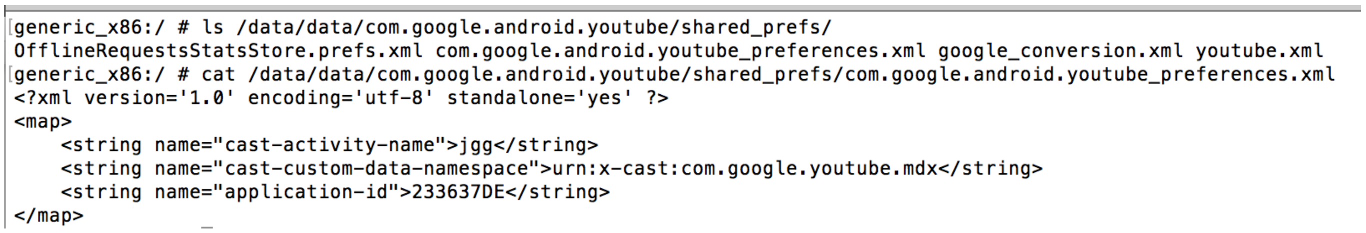
Topic 2: Shared Preferences

Shared Preferences

Shared preferences constitute a “lightweight” means of storing and retrieving key-value pairs (or, more precisely, variable-name / variable-value pairs) of primitive data types (e.g. “fontSize”/12). They persist (i.e., do not disappear) across user sessions and are typically used to store use configuraitons for loading at app start. Examples include default greetings, ringtones, and text font.

Data Format and Data Location

In Android, preferences are stored as XML files under the shared_pref folder in each application folder (/data/data/<apppfullackagename>/). They are private to the application, and thus not accessible by other applications. For example, the YouTube application stores its data under /data/data/com.google.android.youtube. All configuration files are stored in the subdirectory ./shared_prefs using XML files, which essentially store some key-value pairs. The following screenshot shows shared preferences files of the YouTube application.



```
generic_x86:/ # ls /data/data/com.google.android.youtube/shared_prefs/
OfflineRequestsStatsStore.prefs.xml com.google.android.youtube_preferences.xml google_conversion.xml youtube.xml
generic_x86:/ # cat /data/data/com.google.android.youtube/shared_prefs/com.google.android.youtube_preferences.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="cast-activity-name">jgg</string>
    <string name="cast-custom-data-namespace">urn:x-cast:com.google.youtube.mdx</string>
    <string name="application-id">233637DE</string>
</map>
```

APIs to Read and Write Shared Preferences

Android defines a `SharedPreferences` interface for developers to easily save and retrieve persistent key-value pairs of primitive data types in the shared preferences files. The following methods are used to access shared preferences.

Obtain a Reference to the SharedPreferences Object

Before you can save or retrieve data, you need to first obtain a reference to the requested shared preference object using the `getSharedPreferences`() or `getPreferences method`():

```
getSharedPreferences (name:String, mode:Int): SharedPreferences
getPreferences (int mode): SharedPreferences
```

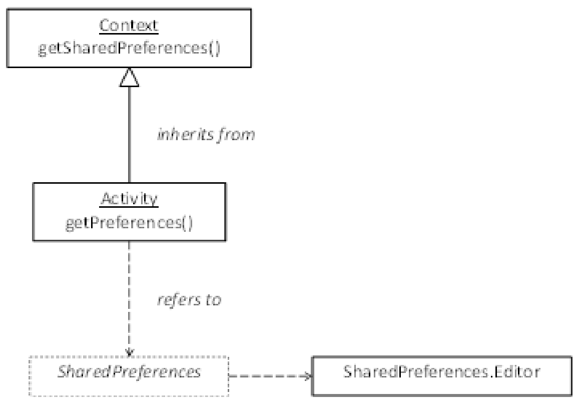
You need to specify the filename and the open mode when you use the `getSharedPreferences`() method. This is useful when there are multiple preference files, each file has a specific name and can be used in different activities.

When using getPreferences() without specifying a file name, the default name is the class name of the activity, from which this method is called. The returned SharedPreferences object is private to this activity.

You can also use another method, getDefaultSharedPreferences(), which is defined in the PreferenceManager: `SharedPreferences getDefaultSharedPreferences (Context context)` The default filename is the <appfullpackagename>_preferences, e.g. “edu.bu.projectportal_preferences.”

All these methods return a SharedPreferences object that is associated with the corresponding preference file. If the file already exists, it simply returns the reference. Otherwise, it will create the file and the object.

This method can only be called in a Context object, such as an activity.



Save Data into the Shared Preferences

SharedPreferences has an inner class Editor that is responsible for saving the data into the shared preferences. Each SharedPreferences object is associated with an Editor object. We will use this to save the initial data into the file or modify the data later.

To get the Editor object, we need to call the edit() method of the SharedPreferences object.

The Editor class defines several setter methods to save different types of data into shared preferences. Each piece of data is a key-value pair, where the key is a string and the value can be different types, such as Boolean, Int, Long, Float, String, etc. (Editor)

Since it is possible that multiple threads try to save values in the shared preferences at the same time, to guarantee this transaction is atomic, we need to use the `commit()`or `apply()` method after editing your shared preferences.

The following code shows how to save the color setting in the shared preference:

```
val sharedPref = getSharedPreferences(“color”,Context.MODE_PRIVATE);
val editor = sharedPref.edit();
editor.putString("color",”RED”);
editor.commit();
```

When this code is added into the activity and run, a file named “color.xml” is generated in the application ./shared_prefs/ subfolder. Here is the file content:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <tring name="color">red</string>
</map>
```

Retrieve Data from the Shared Preferences

To retrieve the data from the shared preferences, we can simply call getter methods of the SharedPreferences object. There are several getter methods to retrieve different types of data, such as getBoolean(), getInt(),getString(), and etc. Two parameters are needed. One is the key which is a string, and the other is the default value if the value is not available in the shared preferences. (SharedPreferences) The following code shows how to retrieve the data from the shared preferences:

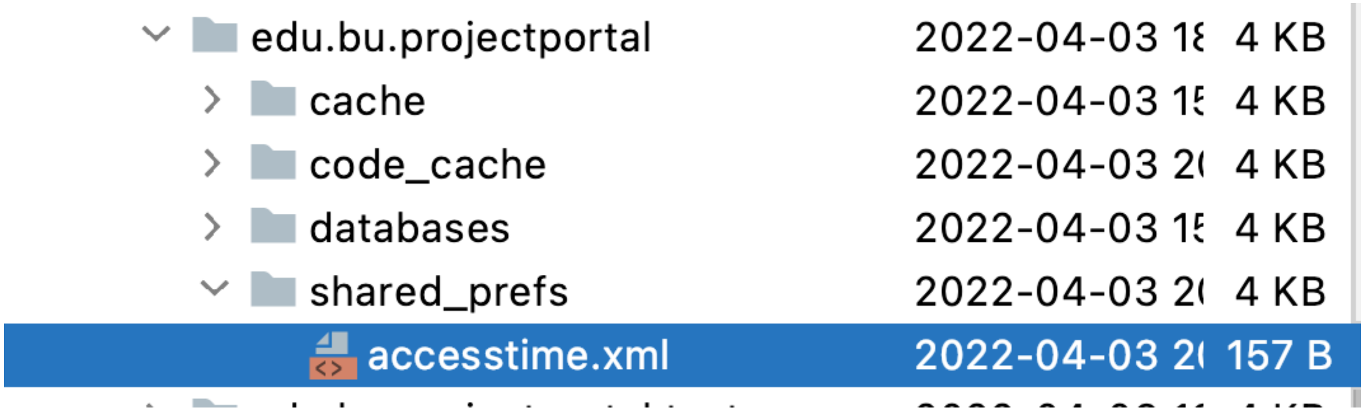
```
val myPref = getSharedPreferences("color", Context.MODE_PRIVATE);
val  colorString = myPref.getString("color", "white"); // the default color is white
```

The Example

Suppose we want to record the last access time across the session in our ProjectPortal project. When the app is opened, it will show the last access time. We can use shared preferences to record this time. Here is the code we add in the main activity.

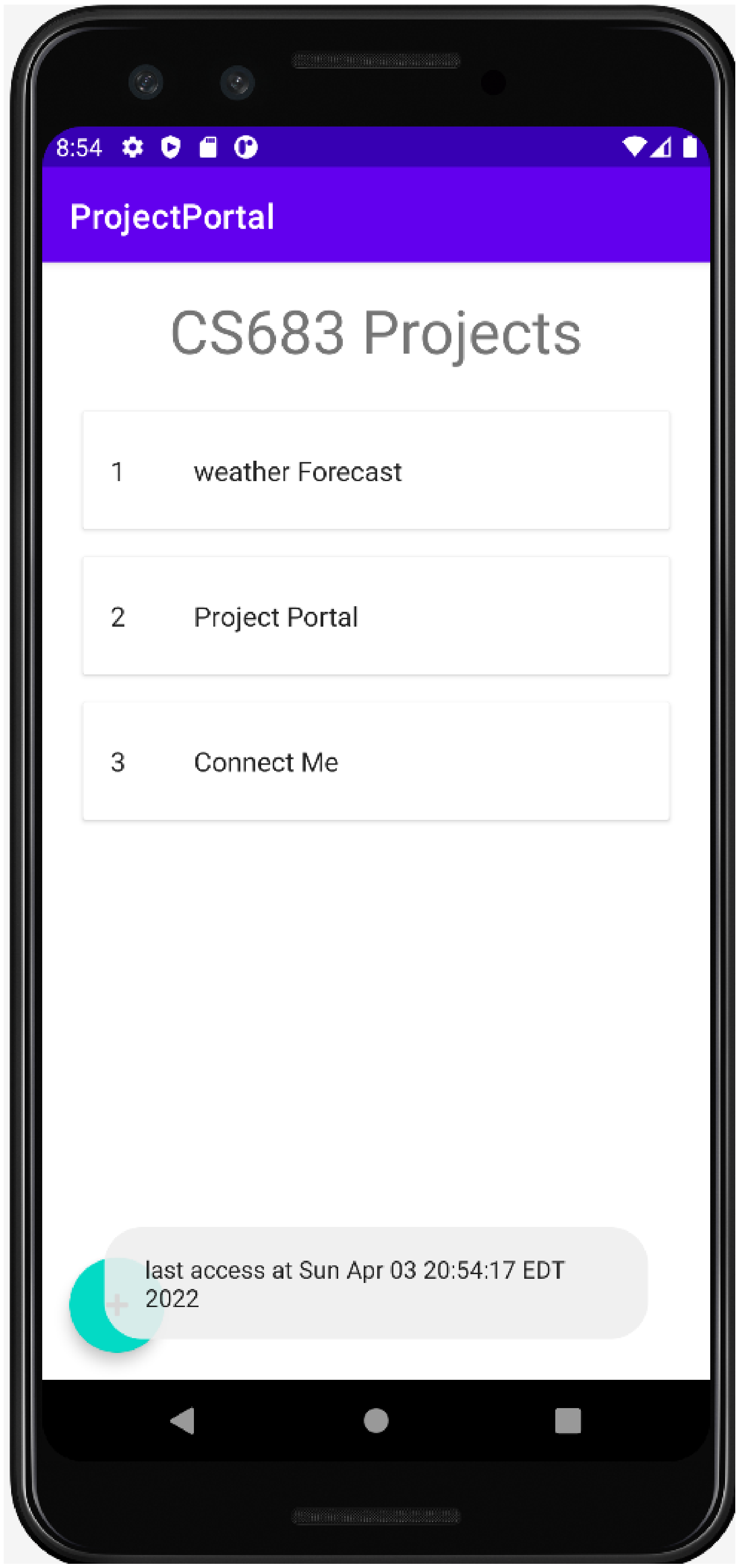
```
fun loadAccessTime():String {
   // get the last access time from the shared preferences. The file name is accesstime.xm
   return getSharedPreferences("accesstime",
       Context.MODE_PRIVATE).getString("lastaccesstime", "")?:""
}
fun saveAccessTime(){
   // get current time and write to the shared preferences file
   val curTime: String = Date().toString()
   getSharedPreferences("accesstime", Context.MODE_PRIVATE).edit().
   putString("lastaccesstime", "last access at $curTime").commit()
}

class MainActivity : AppCompatActivity(){
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.content_slidepane)

       val accessTime =  loadAccessTime()
       Toast.makeText(this, accessTime, Toast.LENGTH_LONG).show()
       saveAccessTime()
  }
. . .
}
```

After executing the application, we can use the “Device File Explorer” window in Android Studio to check the application folder on the device. We can see “accesstime.xml” in the ./shared_prefs subfolder.



```
edu.bu.projectportal      2022-04-03 1  4 KB
  cache                   2022-04-03 1  4 KB
  code_cache              2022-04-03 2  4 KB
  databases               2022-04-03 1  4 KB
  shared_prefs            2022-04-03 2  4 KB
    accesstime.xml        2022-04-03 2  157 B
```

The following screenshot shows the execution result. We can see the last access time shown at the bottom of the screen:


8:54

ProjectPortal

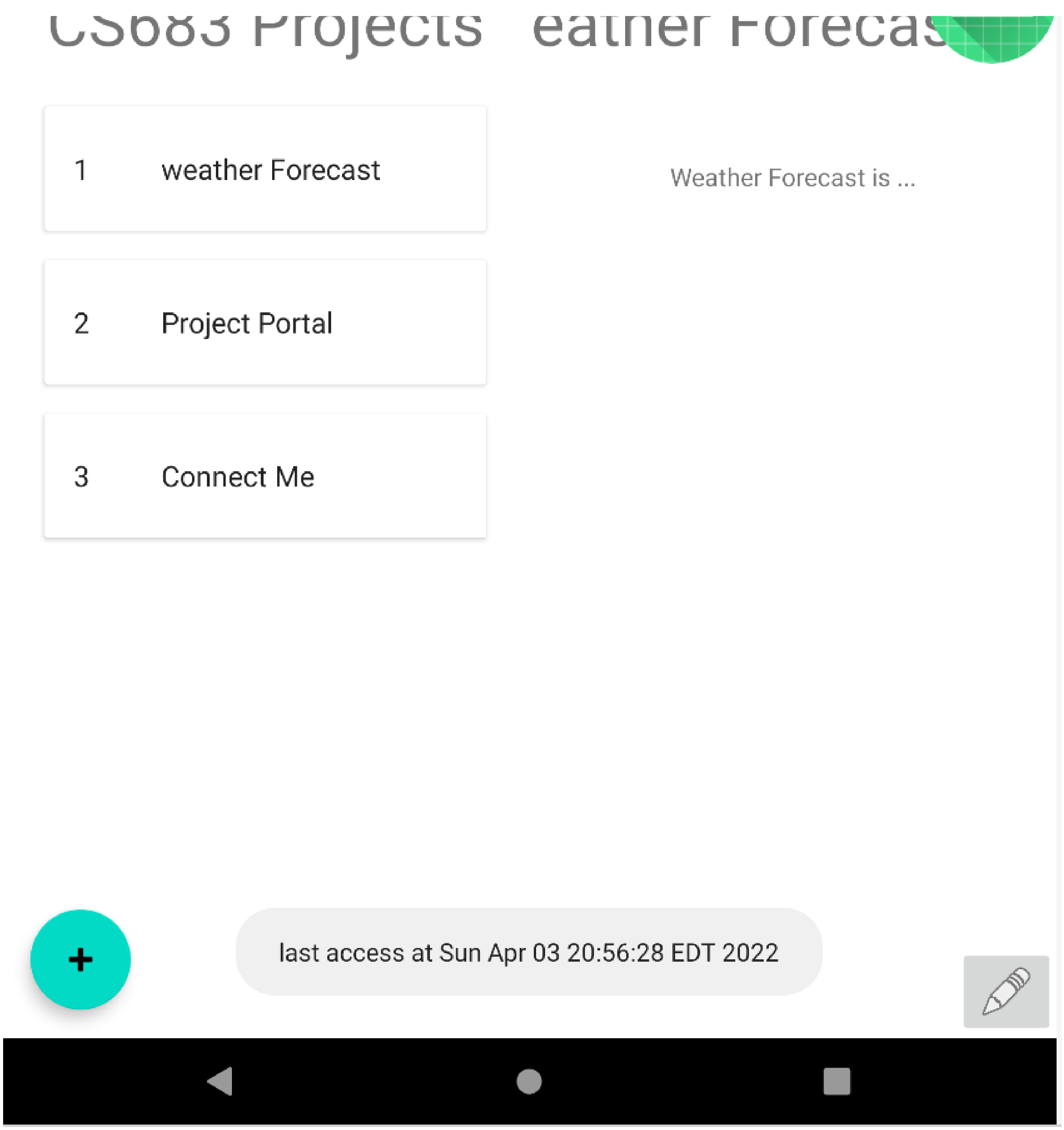
CS683 Projects

1	weather Forecast
2	Project Portal
3	Connect Me

last access at Sun Apr 03 20:54:17 EDT 2022

8:56

ProjectPortal



CS683 Projects eather Forecas

1 weather Forecast

2 Project Portal

3 Connect Me

Weather Forecast is ...

last access at Sun Apr 03 20:56:28 EDT 2022

Test Yourself

When would you use shared preferences to store data?

When would we need to store a small amount of data such as configurations across the sessions?

Test Yourself

Where are the shared preferences stored?

They are stored in a xml file in shared_prefs/ subfolder in the application folder.

Test Yourself

Can the shared preferences be accessed by other applications?

No, it is private to the application.

Test Yourself

Which method is used to obtain a SharedPreferences object?

getSharedPreferences() or getPreferences()

Test Yourself

Complete the following code snippet

```
val sharedPreferences = getSharedPreferences(“Configure”, _____);

_________________

____________.putInt(“Font”. 10);

__________________
```

```
Context.MODE_PRIVATE, val editor  = sharedPreferences.edit(),  editor, edit
```

Topic 3: File Storage

File Storage

Additional data can be stored in files, particularly in large data files, such as images, audio and video files. Files can be stored on the device’s internal flash memory. This is always available as long as there is space. Files can also be stored on external storage units such as SD cards (or emulated SD cards), which may or may not be available.

When using the internal storage, files are inaccessible to other applications by default. They are usually stored under /data/data/<appfullpackagename>/files/. When the application is uninstalled, the files are deleted too. Before Android 7.0 (API level 24), internal files could be made accessible to other apps by means of relaxing file system permissions. However, this is no longer the case in Android 7 and above. If you wish to make the content of a private file accessible to other apps, you may use the FileProvider.

When using the external storage, files are shared by all applications. Therefore, you should only use external storage when no access restriction is required.

We can use the basic file APIs defined in Java/Kotlin to access the file storage. A File object is well-suited for reading or writing large amounts of data in start-to-finish order without skipping around. However, we cannot access files in an arbitrary directory. We can only access files in certain directories. Several APIs can help us choose the right path and manage application files correctly. To release the storage, you should always delete files that are no longer needed.

For more details, please read Android's File Storage webpage.

Internal Storage APIs

File is a class defined in the java.io package. It is a native Java class used to represent files or folders. When a File object is created based on the file path, we can perform basic file operations on the File object. FileInputStream and FileOutputStream are also classes defined in the java.io package. They are used to write and read streams of raw bytes from a file. There are also other classes defined in the java.io package to handle file input/output.

The APIs provided in Android for the file storage are used to specify the predefined path explicitly or implicity. Here is a list of some methods provided:

- getFilesDir(): returns a file representing an internal directory for your app
- getCacheDir(): returns a file representing an internal directory for your app's temporary cache files
- openFileOutput(): returns a file outputstream that is associated with an internal file of your app
- openFileInput(): returns a file inputstream that is associated with an internal file of your app

We can open a file by explicitly specifying the path returned from getFilesDir() in the java file I/O APIs, or we can use `openFileInput()/openFileOuput()` to get the input/output stream of an internal file in the predefined location. Here are two code snippets from the Android developer website:

```
val file = File(context.getFilesDir(), filename)


val filename = "myfile"
val string = "Hello world!"
val outputStream: FileOutputStream
try {
   outputStream = openFileOutput(
       filename,
       MODE_PRIVATE
   )
   outputStream.write(string.toByteArray())
   outputStream.close()
} catch (e: Exception) {
   e.printStackTrace()
}
```

External Storage APIs

Applications can also store data in the external storage in the SD card (or the emulated SD partition). However, you should be aware that the external storage is universally accessible by any other application. Never store anything sensitive in the external storage region. Also, the external storage may not always available. For example, an SD card may be removed by the user. Always verify if the volume is available before accessing it.

We can use the same file APIs in the java.io package to access files. Several APIs provided by Android help get the external storage path and check the external storage status:

- getExternalFilesDir(): returns the absolute path to the directory on the primary shared/external storage device where the application can place persistent files it owns. While files under this directory can be accessed by other applications, they will be deleted if the app is uninstalled. It is usually under /sdcard/Android/data/<appfullpackagename>/files. This is called a private external storage directory.

- getExternalStoragePublicDirectory(): returns a top-level shared/external storage directory for placing files of a particular type. This is where the user will typically place and manage his or her own files. Files under this directory are accessible by other applications. They will NOT be deleted if the app is uninstalled. It is usually under /sdcard/, the root directory of the SD card partition. For example, photos captured by your app or other downloaded files are in this folder. This is called a public external storage directory.
- getExternalStorageState(): returns the current state of the primary shared/external storage media, e.g. if it is mounted or not, if it is for read only.
- getFreeSpace() or getTotalSpace(): gets current available space or the total space in the storage volume, respectively.

```
// Checks if external storage is available for read and write
fun isExternalStorageWritable():Boolean {
   if(Environment.MEDIA_MOUNTED
           .equals(Environment.getExternalStorageState()))
       return true

   return false
}

// Checks if external storage is available to read
fun isExternalStorageReadable():Boolean {
   val state = Environment.getExternalStorageState();
   if (Environment.MEDIA_MOUNTED.equals(state) ||
       Environment.MEDIA_MOUNTED_READ_ONLY.equals(state))
       return true

   return false
}
```

To read or write from or to the external storage, you must request the READ_EXTERNAL_STORAGE/WRITE_EXTERNAL_STORAGE permission in your manifest file. However, beginning with Android 4.4 (API level 19), reading or writing files in your app's private external storage directory (through getExternalFilesDir()) does not require these permissions. Permissions are only required to access files in the public external storage directory. No permissions are needed to read/write to the internal storage:

```
<manifest ...>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" /> ...
</manifest>
```

Test Yourself

What are the differences between the external storage and the internal storage?

Internal: private, not accessible to other applications. External: accessible to other applications

Test Yourself

What are the return path of the following methods called by the package: edu.bu.fileexample

getFilesDir(): _____

getCacheDir(): _____

getExternalFilesDir(): _____

getExtenralCacheDir(): _____

getExternalPublicStorageDir(): _____

/data/data/edu.bu.fileexample/files, /data/data/edu.bu.fileexample/cache, /sdcard/Android/data/edu.bu.fileexample/files, /sdcard/Android/data/edu.bu.fileexample/cache, /sdcard/?

Topic 4: Content Providers

Content Providers

Content providers are the standard interface used to share access to your application data with other applications in Android. If we want to share our application data with other applications, we need to define it as a content provider. Several system applications use content providers to share their data with others. For example, contact data is shared by multiple applications through the contacts provider. It is the source of data in the device's contacts application. We can also access its data in our own application. Another example is the calendar provider. Its data can be shared by multiple applications.

Content providers are important Android components used to provide content to other applications. The underlying storage can be databases or files. Content providers can also handle common data types, as well as video and audio.

In this section, we will learn how to create a content provider and how to access a content provider.

Content Provider Basics

Content Provider APIs

ContentProvider is an abstract class defined in Android. It defines several abstract methods that need to be implemented:

- Provider initialization: onCreate() which is called to initialize the provider
- Provider CRUD Operations:
  - insert(Uri, ContentValues) which inserts new data into the content provider
  - query(Uri, String[],String, String[], String) which returns query data to the caller
  - update(Uri, ContentValues, String, String[]) which updates existing data in the content provider
  - delete(Uri, String, String[]) which deletes data from the content provider
- Provider data type getter: getType(Uri) which returns the MIME (Multi-Purpose Internet Mail Extension) type of data in the content provider

Define Content URIs

So what is a URI? URI stands for Universal Resource Identifier. A content provider is accessed through a content URI. Here is the general form of a content URI `content://authority/path/id` content: the scheme name

authority: the content provider name

path: usually a table name

id: optional, the row number

For example:

content://user_dictionary/words

content://user_dictionary/words/3

An application may have more than one content provider, and a single content provider may provide access to multiple forms of content. Using URI we can specify a unique resource to be accessed. Android defines an abstract class, Uri, to represent an immutable URI reference.

FileProvider and DocumentsProvider are two special subclasses of ContentProvider. A file provider facilitates secure sharing of files associated with an app by creating a content://Uri for a file instead of a file:///Uri. A document provider offers read and write access to durable files, such as files stored on a local disk, or files in a cloud storage service.

Accessing Content Providers through ContentResolver

We cannot access the ContentProvider objects directly; instead we need to access them through a ContentResolver object. ContentResolver is an abstract class defined by Android. To get a ContentResolver instance for our application's package, we call the `getContentResolver`()method from the **context** of the app. Then, we can call its CRUD methods on the persistent storage. Based on the Uri passed in the method, it will call the identically-named method in the provider object specified using the provider's URI.

A lot of data on the Android devices are provided to applications using content providers. We can use contentResolver to access those data. For example, we can access media files such as images, videos, audios using a ContentResolver object with the specified media file URIs. The following code shows how to get a collection of image files with a name similar to "project". Suppose the following code is in the fragment class, so we can get the context from its activity, and then its contentResolver. The URI for image files is `MediaStore.Images.Media.EXTERNAL_CONTENT_URI`, which include photos and screenshots that are stored in the DCIM/ and Pictures/ directories. The system adds these files to the **MediaStore.Images** table. In the following query, we need to provide a number of parameters. The project is an array of field names that should return from the query. The selection is the sql query statement with potential arguments specified as "?". The selectionArgs is an array of parameters needed to specify for the query, and the sortOrder specifies the sorting order of returned records. The return result from a query is a cursor that accesses the return results from a query. We can iterate through the cursor and get the value of each field from each record of returned query results.

```
val projection = arrayOf(MediaStore.Images.Media._ID,
   MediaStore.Images.Media.DISPLAY_NAME,
   MediaStore.Images.Media.SIZE)
val selection = "${MediaStore.Images.Media.DISPLAY_NAME} like ?"
val selectionArgs =arrayOf("project")
val sortOrder = "${MediaStore.Images.Media.DISPLAY_NAME} ASC"
activity?.contentResolver?.query(
   MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
   projection, selection, selectionArgs, sortOrder)?.use { cursor ->
   val idColumn = cursor.getColumnIndexOrThrow(MediaStore.Images.Media._ID)
   val nameColumn =
       cursor.getColumnIndexOrThrow(MediaStore.Video.Media.DISPLAY_NAME)
   val sizeColumn = cursor.getColumnIndexOrThrow(MediaStore.Video.Media.SIZE)
```

```
    while (cursor.moveToNext()) {
        // Get values of columns for a given image.
        val id = cursor.getLong(idColumn)
        val name = cursor.getString(nameColumn)
        val size = cursor.getInt(sizeColumn)

        val contentUri: Uri = ContentUris.withAppendedId(
            MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
            id
        )
        //
        imageList += myImage(contentUri, name, size)
    }
}
```

It is very convenient to use ContentResolver to access the content provider provided by other applications. As long as we know the Uri and the column names, we can access them directly. Android ships with a number of content providers that store common data such as contact, calendar, SMS/MMS beside media files. You can find the classes at android.provider. In the same way shown above, we can use a content resolver to access the data in those content providers.

Request Necessary Permissions

For most of those content providers, we may need to request **read access permissions** first in the **manifest file** in order to access them, using the <uses-permission> element and the exact permission name defined by the provider. When users install your application, they implicitly grant this request. The exact name of the read access permission for the provider can be found in the provider's documentation. For example, you need to request the READ_CONTACTS permission to access the contact provider by adding the following line in the AndroidManifest.xml, under the application tag.

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

You may or may not need to request permission to access media files, depending on the Android version and the type of data you want to access. For more details, please read Access media files from shared storage.

Create a Content Provider

Subclassing ContentProvider

To create a content provider, we need to first subclass ContentProvider and implement all its abstract methods. If the underlying storage is a database, we can implement the above CRUD methods using sqlite database CRUD methods. Actually, we can see that both APIs are quite similar and well-mapped. The main difference is the first parameter. The content provider CRUD method needs to specify a Uri, while the database CRUD methods need to specify a table. If we implement the database using SQLitedatabase APIs directly, it is pretty easy to implement those content provider CRUD methods. If we use Room database or other APIs, it is a little bit more complicated. Since most of our projects will not create a content provider, instead just using the content providers provided by the Android system, we will only discuss this at the conceptual level.

One key step to implement content provider CRUD methods based on the database CRUD methods is to match Uri with database tables. We can use UriMatcher to aid in matching URIs in content providers. UriMatcher is a utility class to aid in matching URIs in content providers. To use this class, we need to first build up a tree of `UriMatcher` objects. Use the following two method to build the tree:

- UriMatcher(int code): the public constructor method to create the root node of the URI tree void addURI(String authority, String path, int code): adds a URI to match and the code to return when this URI is matched

To match the Uri, we can call the match() method.

- int match(Uri uri): Tries to match against the path in a url, returns the match code

Content Provider APIs	SQLiteDatabase APIs
insert(Uri, ContentValues) which inserts new data into the content provider query(Uri, String[],String, String[], String) which returns query data to the caller update(Uri, ContentValues, String, String[]) which updates existing data in the content provider delete(Uri, String, String[]) which deletes data from the content provide	insert(tableName, ContentValues) which inserts new data into the sqlite database query(tableName, String[],String, String[], String) which returns query data to the caller update(TableName, ContentValues, String, String[]) which updates existing data in the the sqlite database delete(tableName, String, String[]) which deletes data from the the sqlite database

Declare the ContentProvider in the AndroidManifest.xml

Since the purpose of the content provider is to share the data with other applications, we need to define the content provider components in the AndroidManifest.xml file using the <provider> element. Attributes include:

- android:authority: the full authority URI of the content provider
- android:name: the name of the class that implements the content provider, usually the same value as the authority.
- android:permission (or android:readPermission, android:writePermission): permissions that must be held by client applications to get access to the underlying data. If no permissions are declared, the default behavior is for permission to be allowed for all applications in devices with API 16 or under, but *NOT allowed in a device with API 17.* Permissions can be set to cover the entire content provider or to limit access to specific tables and records using the <path-permission> tag.

For example, we can add the following code into the manifest file to define a provider for the ProjectPortal app. Here we use a customized permission “edu.bu.projectportal.read”.

```
<provider
   android:name = ".ProjectPortalContentProvider"
   android:authorities="edu.bu.projectportal.ProjectPortalContentProvider"
   android:exported= "true"
   android:permission="edu.bu.projectportal.read">
</provider>
```

Test Yourself

Briefly describe steps used to create a content provider.

Test Yourself

What does a Content URI consist of?

scheme name “content://”, provider authority name, and path

Test Yourself

How would you access a content provider?

Conclusion

In this module, we have discussed how to store data in our app. We discussed several different types of storage, namely , databases, shared preferences, internal or external file storage, and content providers. Each type has its own features and purpose. Most structured application data are stored in the SQLite database. Through examples, we learned various Android APIs and how to use these APIs to manage different types of data.