Mobile Application Development

Week11　Use Room Database, DAO, @Entity

Yi SUN
Kobe Institute of Computing

Use the SQLite and Room to manage the data in Local storage

SQLite is a free and open-source relational database engine written in the C programming language. It is not a standalone app; rather, it is a library that software developers embed in their apps. As such, it belongs to the family of embedded databases. It is the most widely deployed database engine, as it is used by several of the top web browsers, operating systems, mobile phones, and other embedded systems.

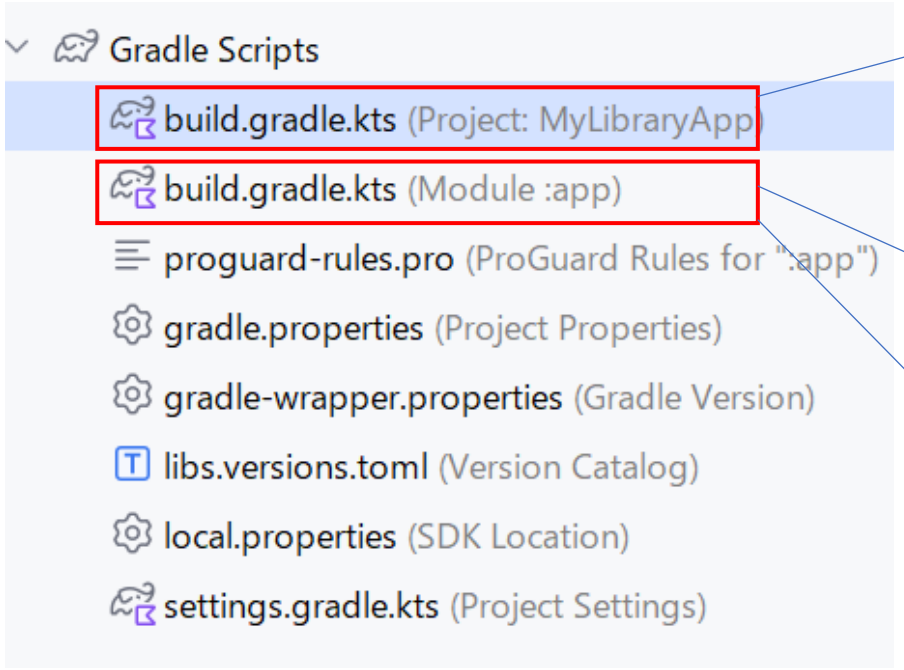
The ***Room*** persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.

https://en.wikipedia.org/wiki/SQLite

https://developer.android.com/jetpack/androidx/releases/room

Add the navigation dependencies

- Add the dependency in your build.gradle files
- (project-level) and (app-level)




```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
    alias(libs.plugins.kotlin.compose) apply false
    val room_version = "2.7.1"
    id("androidx.room") version "$room_version" apply false
    id("com.google.devtools.ksp") version "2.1.20-2.0.0" apply false
}
```

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.compose)
    id("androidx.room")
    id("com.google.devtools.ksp")
}
```

```
dependencies {

    val room_version = "2.7.1"
    implementation("androidx.room:room-runtime:$room_version")
    ksp("androidx.room:room-compiler:$room_version")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
```

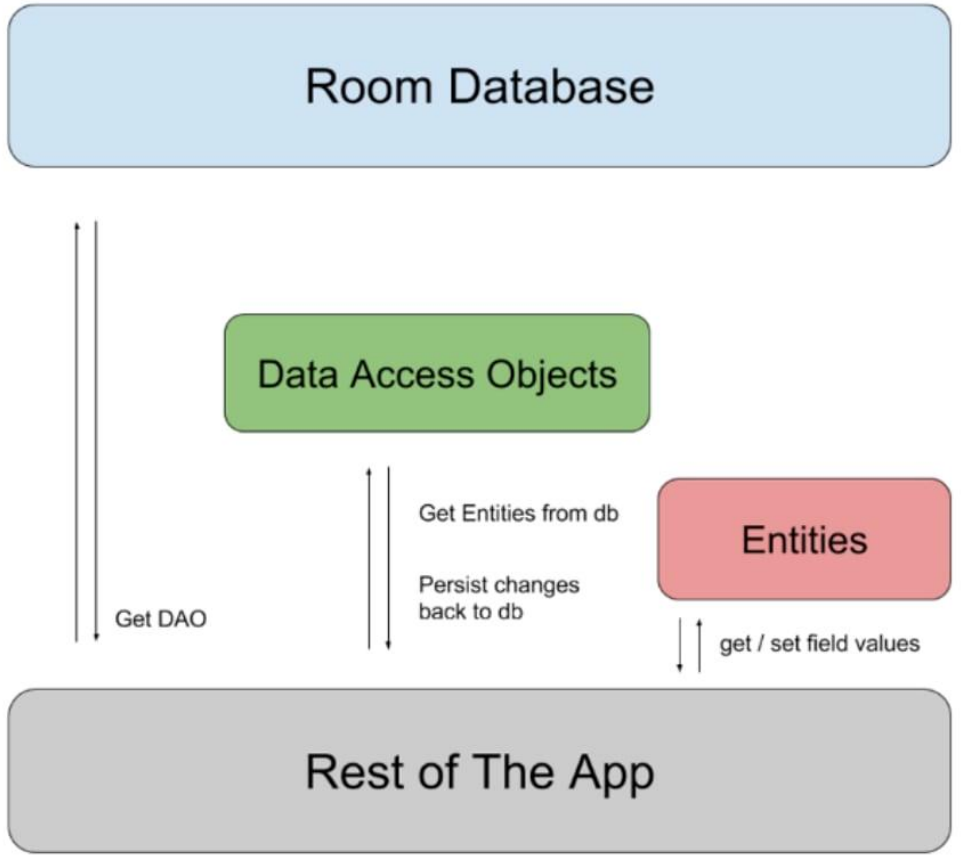
Change the data class to data Entity

Book.kt

```
@Entity(tableName = "book-table")
data class Book(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0L,
    @ColumnInfo(name="book-title")
    val title: String="",
    @ColumnInfo(name="book-isbn")
    val isbn: String="",
    @ColumnInfo(name="book-author")
    val author: String=""
)
```

Accessing data using Room DAOs

- When you use the Room persistence library to store your app's data, you interact with the stored data by defining data access objects, or DAOs. Each DAO includes methods that offer abstract access to your app's database. At compile time, Room automatically generates implementations of the DAOs that you define.
- By using DAOs to access your app's database instead of query builders or direct queries, you can preserve separation of concerns, a critical architectural principle. DAOs also make it easier for you to mock database access when you test your app.



https://developer.android.com/training/data-storage/room/accessing-data

Accessing data using Room DAOs

- Room Entity
  - Define an entity to represent the object to be saved
  - Each entity corresponds to a table in the associated Room database, and each instance of the entity represents a data row in the corresponding table
- Room DAO
  - Define a Data Access Object (DAO) to operate on the data to be saved
  - Each DAO has methods that enable abstract access to the application's database
  - Automatically generates the implementation of the DAO defined at compile time
- Room Database class
  - Serves as the main access point for basic connections to the application's persistent data, holding the database

Create the BookDao file

BookDao.kt

```
@Dao
abstract class BookDao {

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    abstract fun addBook(bookEntity: Book)

    @Query("select * from `book-table`")
    abstract fun getAllBook(): Flow<List<Book>>

    @Update
    abstract suspend fun updateBook(bookEntity: Book)

    @Delete
    abstract suspend fun deleteBook(bookEntity: Book)

    @Query("Select * from `book-table` where id=:id")
    abstract fun getBookById(id:Long): Flow<Book>
}
```

Create a database file and register the Dao

BookDatabase.kt

```
@Database(
    entities = [Book::class],
    version = 1,
    exportSchema = false
)

abstract class BookDatabase : RoomDatabase() {
    abstract fun bookDao(): BookDao
}
```

https://developer.android.com/training/data-storage/room

Create a Repository to register the function from Dao

BookRespository.kt

```
class BookRespository(private val bookDao: BookDao) {
    suspend fun addBook(book:Book){
        bookDao.addBook(book)
    }

    fun getBooks(): Flow<List<Book>> = bookDao.getAllBook()

    fun getBookById(id:Long) :Flow<Book> {
        return bookDao.getBookById(id)
    }

    suspend fun updateBook(book:Book) {
        bookDao.updateBook(book)
    }

    suspend fun deleteBook(book: Book){
        bookDao.deleteBook(book)
    }
}
```

Use Dispatchers and lateinit Var in ViewModel

BookViewModel.kt

```
class BookViewModel(
    private val bookRepository: BookRespository
): ViewModel() {
    var bookTitleState by mutableStateOf("")
...
    lateinit var getAllBooks: Flow<List<Book>>
    init {
        viewModelScope.launch {
            getAllBooks = bookRepository.getBooks()
        }
    }
    fun getBookById(id:Long):Flow<Book>{
        return bookRepository.getBookById(id)
    }
    fun addBook(book: Book){
        viewModelScope.launch(Dispatchers.IO) {
            bookRepository.addBook(book=book)
        }
    }
    fun updateBook(book: Book){
        viewModelScope.launch(Dispatchers.IO) {
            bookRepository.updateBook(book=book)
        }
    }
    fun deleteBook(book: Book){
        viewModelScope.launch(Dispatchers.IO) {
            bookRepository.deleteBook(book=book)
        }
    }
```

lateinit var is a Kotlin modifier that allows you to declare non-null properties without initializing them at declaration time. It essentially tells the compiler: "I promise to initialize this property before using it."
In the example, getAllBooks is declared as a non-null Flow<List<Book>> that will be initialized sometime after the BookViewModel is created.

Dispatchers.IO is specifically optimized for I/O-bound tasks such as: 1,Reading or writing files. 2, Making network requests. 3, Querying databases. 4, Other blocking I/O operations.

These methods provide the CRUD (Create, Read, Update, Delete) operations for the application:

Use Graph to initialize the database

Create a new file of Graph object

BookDatabase.kt

```
object Graph {
    lateinit var database: BookDatabase

    val bookRespository by lazy {
        BookRespository(bookDao = database.bookDao())
    }

    fun provide(context: Context){
        database = Room.databaseBuilder(context,BookDatabase::class.java,"booklist.db").build()
    }
}
```

This creates a lazy property for bookRespository. It means bookRespository won't be created until it's first accessed.It creates an instance of BookRespository and injects bookDao from the BookDatabase.

This function initializes the database using Room's databaseBuilder.It tells Room to build a BookDatabase object using the given context and the name "booklist.db" for the SQLite file.

The Graph object in this code represents a simple dependency injection container (also called a service locator) that manages database-related dependencies throughout the Android application. The Graph singleton serves several important functions:
1, Dependency Container: It acts as a central location for creating, storing, and providing access to application dependencies (database and repository instances). 2, Database Provider: It initializes and provides access to the Room database throughout the app, ensuring only one database instance exists. 3, Repository Provider: It creates and provides the BookRepository instance that application components can use to interact with data.

Add the Application Class and use the Graph object

Add a new Class extend from Application class, and register the application to the AndroidManifest.xml file

BookListApp.kt

```
class BookListApp:Application() {
    override fun onCreate() {
        super.onCreate()
        Graph.provide(this)
    }
}
```

AndroidManifest.xml

```
<application
    android:name=".BookListApp"
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
```

Use the Graph repository in BookViewModel

Graph.kt

```
class BookViewModel(
    private val bookRepository: BookRespository = Graph.bookRespository
): ViewModel() {

    var bookTitleState by mutableStateOf("")
    var bookAuthorState by mutableStateOf("")

...
```

Add Book action and using Snackbars to show the status info

AddEditDetailView.kt

```
fun AddEditDetailView(
    id: Long,
    viewModel: BookViewModel,
    navController: NavController
){
    val snackMessage = remember{
        mutableStateOf("")
    }
    val scope = rememberCoroutineScope()
    val snackbarHostState = remember { SnackbarHostState() }

    Scaffold(
        topBar = {
            AppBarView(title = if(id != 0L) "Update Book info" else "Add New
Book")
            {navController.navigateUp()}
        },
        snackbarHost = { SnackbarHost(snackbarHostState) }
    )
        {
```

AddEditDetailView.kt

```
Button(onClick = {
    if(viewModel.bookTitleState.isNotEmpty()&&
        viewModel.bookAuthorState.isNotEmpty()&&
        viewModel.bookIsbnState.isNotEmpty()){
        if(id !=0L){

        }else {
            viewModel.addBook(
                Book(
                    title = viewModel.bookTitleState.trim(),
                    isbn = viewModel.bookIsbnState.trim(),
                    author = viewModel.bookAuthorState.trim()
                )
            )
            snackMessage.value = "Book has been recorded"

        }
    }else{
        snackMessage.value = "Enter fields to record a Book."
    }
    scope.launch {
        snackbarHostState.showSnackbar(snackMessage.value)
        navController.navigateUp()
    }
}){
```

Read the data from database and display

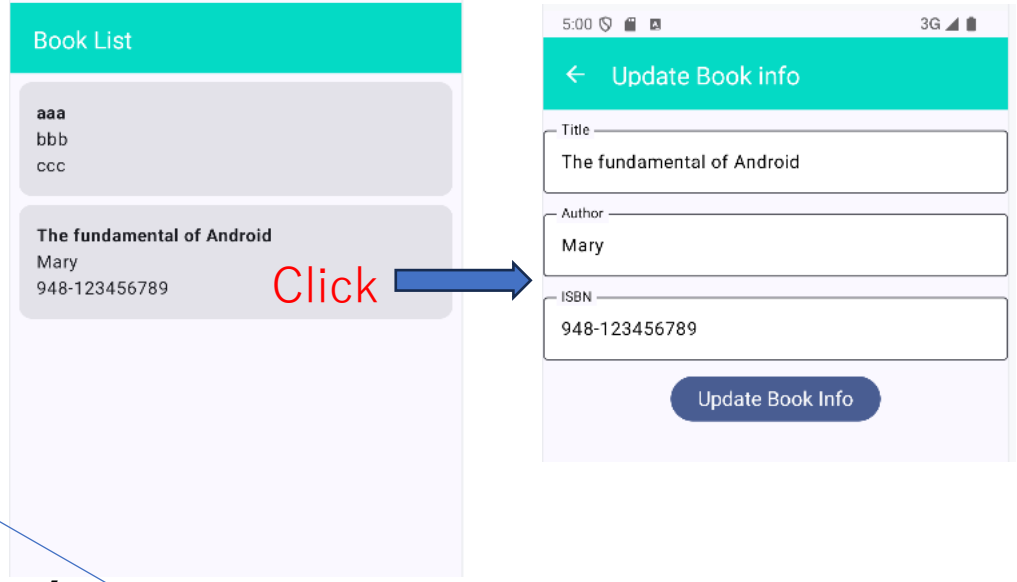
HomeView.kt

```
@Composable
fun HomeView(modifier: Modifier,
             navController: NavController,
             viewModel: BookViewModel
){
    Scaffold(
        modifier = modifier,
        topBar ={ AppBarView(title="Book List") },
        floatingActionButton = {
            FloatingActionButton(
                onClick = {navController.navigate(Screen.AddScreen.route)},
                modifier = Modifier.padding(all = 20.dp),
            ) {
                Icon(
                    imageVector = Icons.Default.Add, contentDescription = "add button"
                )
            }
        }
    ) {
        val booklist = viewModel.getAllBooks.collectAsState(initial = listOf())
        LazyColumn(modifier = Modifier.fillMaxSize().padding(it)) {
            items(booklist.value){
                book -> BookItem(book = book) { }
            }
        }
    }
}
```

Attach the action for the card, move it to the edit view.

HomeView.kt

```
Scaffold(
    modifier = modifier,
    topBar ={ AppBarView(title="Book List") },
    floatingActionButton = {
        FloatingActionButton(
            onClick = {navController.navigate(Screen.AddScreen.route + "/0L")},
            modifier = Modifier.padding(all = 20.dp),
        ) {
            Icon(
                imageVector = Icons.Default.Add, contentDescription = "add button"
            )
        }
    }
) {
    val booklist = viewModel.getAllBooks.collectAsState(initial = listOf())
    LazyColumn(modifier = Modifier.fillMaxSize().padding(it)) {
        items(booklist.value){
            book -> BookItem(book = book) {
                val id = book.id
                navController.navigate(Screen.AddScreen.route + "/$id")
            }
        }
    }
```



Navigation.kt

```
composable(Screen.HomeScreen.route){ HomeView(modifier,navController,viewModel)}
    composable(Screen.AddScreen.route + "/{id}",
        arguments = listOf(
            navArgument("id"){
                type = NavType.LongType
                defaultValue = 0L
                nullable = false
            }
        )
        ){entry->
        val id = if(entry.arguments != null) entry.arguments!!.getLong("id") else 0L
        AddEditDetailView(id = id, viewModel = viewModel, navController=navController )
    }
}
```

Attach the action for the card, move it to the edit view.

AddEditDetailView.kt

```
fun AddEditDetailView(
    id: Long,
    viewModel: BookViewModel,
    navController: NavController
){
    val snackMessage = remember{
        mutableStateOf("")
    }
    val scope = rememberCoroutineScope()
    val snackbarHostState = remember { SnackbarHostState() }
    if (id != 0L){
        val book = viewModel.getBookById(id).collectAsState(initial = Book(0L,"","",""))
        viewModel.bookTitleState = book.value.title
        viewModel.bookIsbnState = book.value.isbn
        viewModel.bookAuthorState = book.value.author
    }else{
        viewModel.bookTitleState = ""
        viewModel.bookIsbnState = ""
        viewModel.bookAuthorState = ""
    }


    Scaffold(
        topBar = {
```

Update the Existing Book info

AddEditDetailView.kt

```
Button(onClick = {
    if(viewModel.bookTitleState.isNotEmpty()&&
        viewModel.bookAuthorState.isNotEmpty()&&
        viewModel.bookIsbnState.isNotEmpty()){
        if(id !=0L){
            viewModel.updateBook(
                Book(
                    id = id,
                    title = viewModel.bookTitleState.trim(),
                    isbn = viewModel.bookIsbnState.trim(),
                    author = viewModel.bookAuthorState.trim()
                )
            )
            snackMessage.value = "Book info has been updated"
        }else {
            viewModel.addBook(
                Book(
                    title = viewModel.bookTitleState.trim(),
                    isbn = viewModel.bookIsbnState.trim(),
…
```

Swipe to delete the data card

HomeView.kt

```
LazyColumn(modifier = Modifier.fillMaxSize().padding(it)) {
    items(booklist.value){
        book ->
        var dismissState = rememberSwipeToDismissBoxState(
            confirmValueChange = { dismissValue ->
                when (dismissValue){
                    SwipeToDismissBoxValue.EndToStart -> {
                        viewModel.deleteBook(book)
                        true
                    }
                    else-> false
                }
            }
        )
```

dismissState to hold swipe state.

cinfirmValueChange is called when swipe reaches threshold.

If swiped End→Start (i.e., right-to-left), it calls viewModel.deleteBook(book) and returns true (which triggers the dismissal animation).

Renders a background (the red delete icon) behind the item as you swipe.Only allows swiping from end to start (enableDismissFromEndToStart = true).backgroundContent

A full-size Card with a right-aligned red trash icon that appears as you swipe.

```
SwipeToDismissBox(
    state = dismissState,
    enableDismissFromStartToEnd = false,
    enableDismissFromEndToStart = true,
    backgroundContent = {
        Card(
            modifier = Modifier
                .fillMaxSize()
                .padding(8.dp)
        ) {
            Column(
                modifier = Modifier.fillMaxSize(),
                horizontalAlignment = Alignment.End,
                verticalArrangement = Arrangement.Center
            ) {
                Icon(
                    imageVector = Icons.Default.Delete,
                    contentDescription = "Delete",
                    tint = Color.Red,
                    modifier = Modifier
                        .padding(end = 24.dp)
                                    .size(36.dp)
                                )
            }
        }
    }
) {
    BookItem(book = book){
        val id = book.id
        navController.navigate(Screen.AddScreen.route + "/$id")
    }
}
```