COMP-304 Winter 2023

Review of Lecture 7

- **Kotlin Coroutines**
  - Android uses Kotlin coroutines to manage long-running tasks
  - A coroutine is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously
    - Lightweight
    - Fewer memory leaks
    - Built-in cancellation support
    - Jetpack integration
- **Flow**
  - **A flow is a type that can emit multiple values sequentially**
  - conceptually a stream of data that can be computed **asynchronously**
  - Flows are **built on top of coroutines** and can provide multiple values

Review of Lecture 7

- **Connecting to Internet resources**
  - Retrofit - the de facto official way to communicate with an HTTP API on Android
  - Retrofit is an open-source library created and maintained by Square (square.github.io/retrofit).
  - meant to define the contracts for many different types of network requests
  - you write an interface with annotated instance methods, and Retrofit creates the implementation
  - uses **OkHttp**, to handle making an HTTP request and parsing the HTTP response.
- **Parsing internet resources**
  - Android includes the standard **org.json** package, which has classes that provide access to creating and parsing JSON text (such as JSONObject and JSONArray).
  - Alternatively, use **Moshi** (github.com/square/moshi) another library from Square.
  - Moshi maps JSON data to Kotlin objects automatically.
  - **Coil** is used for efficient image loading
  - **Coil** leverages all the convenient features of the modern Kotlin language and integrates seamlessly with coroutines to manage performing work in the background.

Creating and Using Databases in Android Apps

Objectives:

- Build Android apps using **Room** persistence library to **add**, **modify**, and **delete** saved data
- Query Room databases and **observe query result changes** using **Flow**.

Structured Data Storage in Android

- **SQLite** is an **open-source** standards-compliant, lightweight, relational database
  - to create fully encapsulated **relational databases** for your applications, and use them to store and manage complex, structured application data.
- Android applications store their databases (SQLite or otherwise) in a special application directory:
  - With emulator running, select View/Tool Windows/Device File Explorer, go to:

/data/data/<application package name>/**databases**/<databasename>

- The **Room** persistence library creates and maintains this database for you.

Structured Data Storage in Android

- **Room persistence library** - an abstraction layer over SQLite that allows you to persist your applications data using the powerful SQLite database.

- **Firebase Realtime NoSQL Database** – to create and use a cloud-hosted NoSQL database.

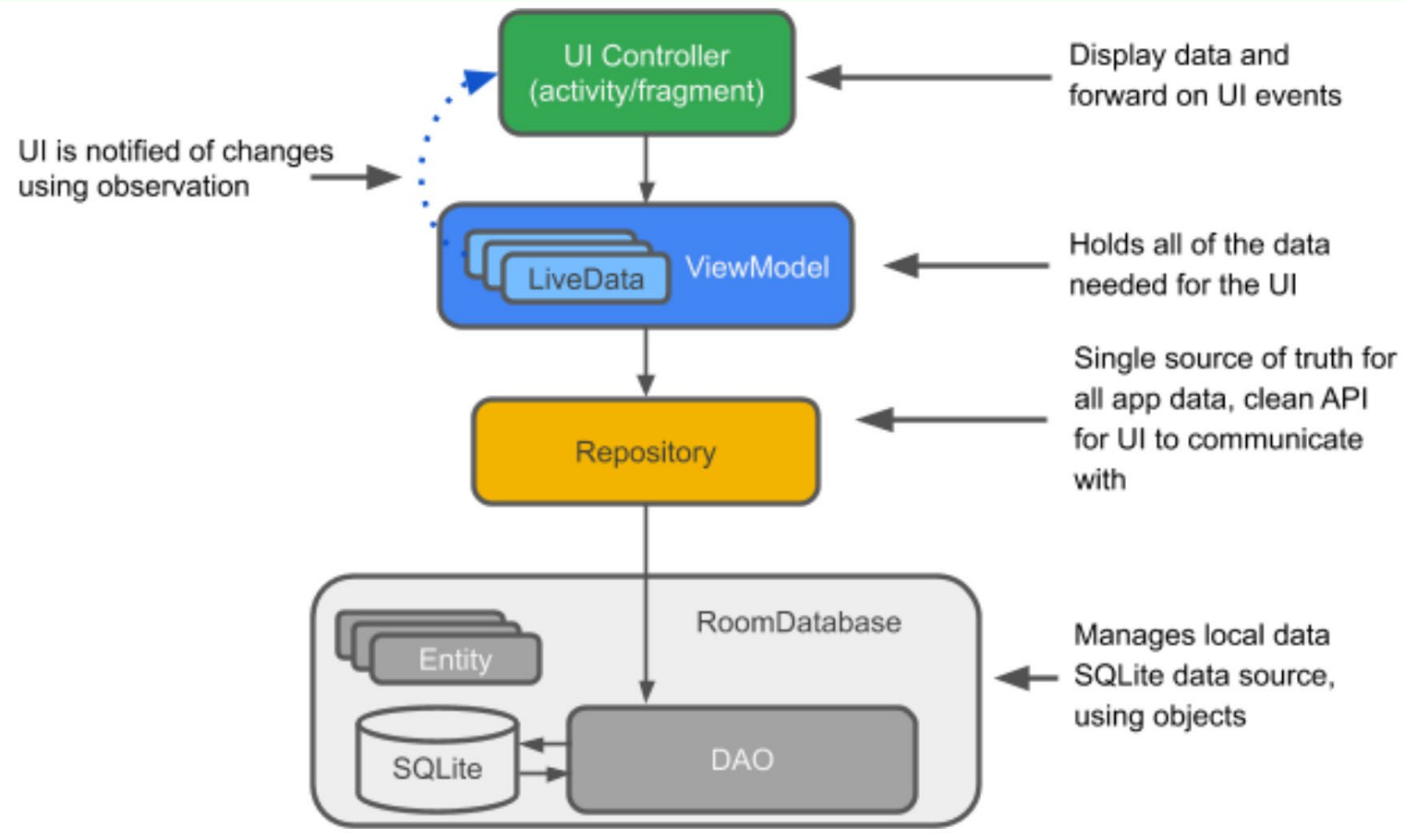
MVVM—Model View ViewModel.

- MVVM is an architectural pattern which **enhances the separations of concerns** by separating UI from business logic. It's composed of the following layers:

1. **Model**
   - represents the **data and database functionalities**
2. **ViewModel**
   - **interacts with model** and also prepares observable(s) that can be observed by a View
   - uses a LiveData or **Flow** object to receive live updates
3. **View**
   - **observes** (or subscribe to) **a ViewModel observable** to get data in order to update UI elements accordingly

- Makes **testing easier**

Architecture Components



Architecture Components

- **Entity**: When working with Architecture Components, this is an **annotated class that describes a database table**.

- **DAO**: Data access object – a **mapping of SQL queries to functions**.
  - Contains the methods used for accessing the database
  - In SQLite you have to define these in your SQLiteOpenHelper class.
  - When you use a DAO, you call the methods, and Room takes care of the rest.

- **Repository**: A class that you create for **managing multiple data sources**.

Architecture Components

- **ViewModel -** provides data to the UI and acts as a communication center **between the Repository and the UI**.
  - Hides where the data originates from the UI.
  - ViewModel instances **survive configuration changes**.

- **LiveData**: A **data holder** class that **can be observed**.
  - Always **holds/caches** latest version of data.
  - **Notifies** its observers when the data has changed.
  - LiveData is **lifecycle aware**.
  - UI components just observe relevant data and don't stop or resume observation.
- We will use **Flow** instead of LiveData:
  - Flow is built on top of Kotlin Coroutines that can handle streams of values, and transform data in complex multi-threaded ways.

Object Relational Mapping (ORM)

- **Room maps** database **objects** to Java **objects:**
  - maps class **variables** to table **columns**
  - maps **methods** to SQL **statements**

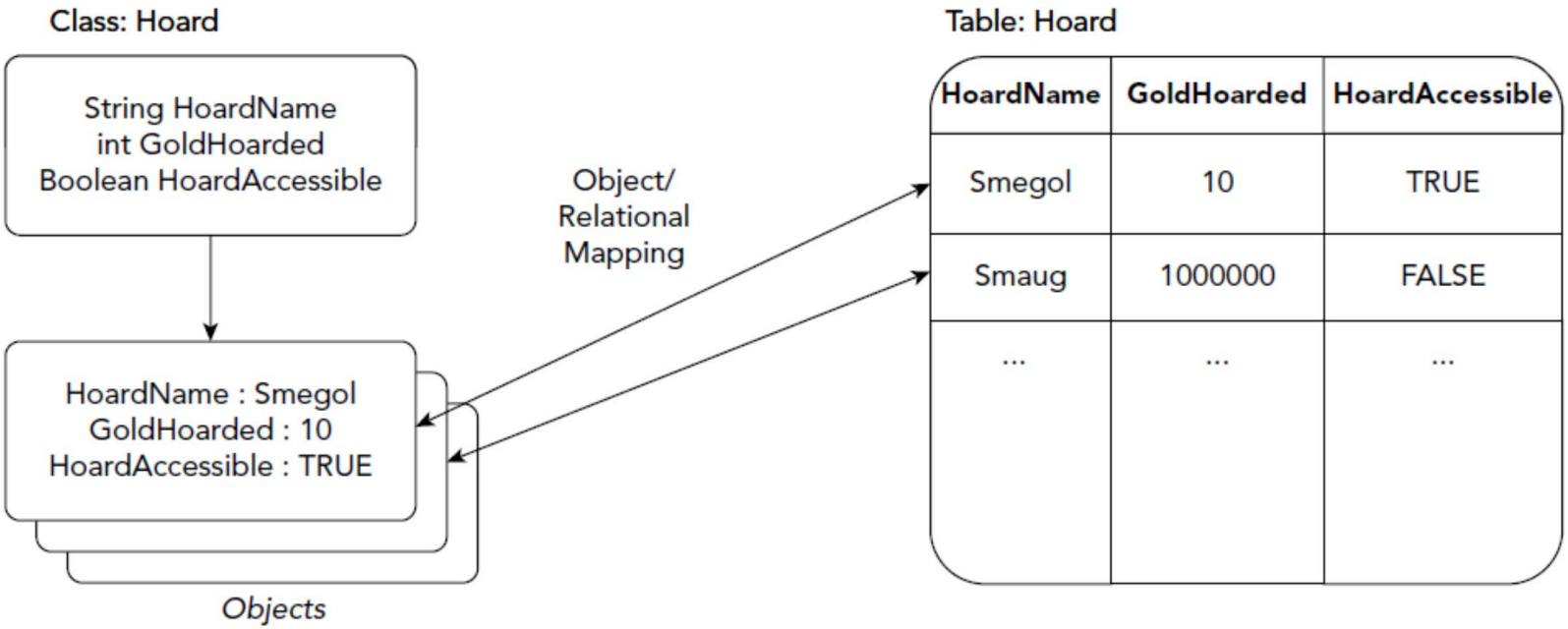


FIGURE 9-1

Room Persistence Library

- Room simplifies this by allowing you to **use annotations within your class definitions**.
- Open your app module build.gradle file and **add the** following **Room library dependencies** within the dependencies node (as always, you should indicate the newest version available to you):

```
dependencies {
[... Existing dependencies ...]
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1“
    implementation "androidx.room:room-runtime:2.4.3"
    kapt "androidx.room:room-compiler:2.4.3“
   // optional - Kotlin Extensions and Coroutines support for Room
   implementation "androidx.room:room-ktx:2.4.3“
   }
```

Defining a Room Database

- The Room persistence model requires you to define **three components**:

1. **Entity -** one or more classes, annotated with the @Entity annotation, which define the **structure of a database table** that will be used to store instances of the annotated class.
2. **Data Access Object -** a class annotated with the @Dao annotation that will **define the methods** used to modify or query the database.
3. **Room Database -** an abstract class annotated with the @Database annotation that extends RoomDatabase.
   - This class is the **main access point** for the underlying SQLite connection
   - must also **include an abstract method** that returns the Data Access Object class and the list of entities the database will contain.

Defining a Room Database

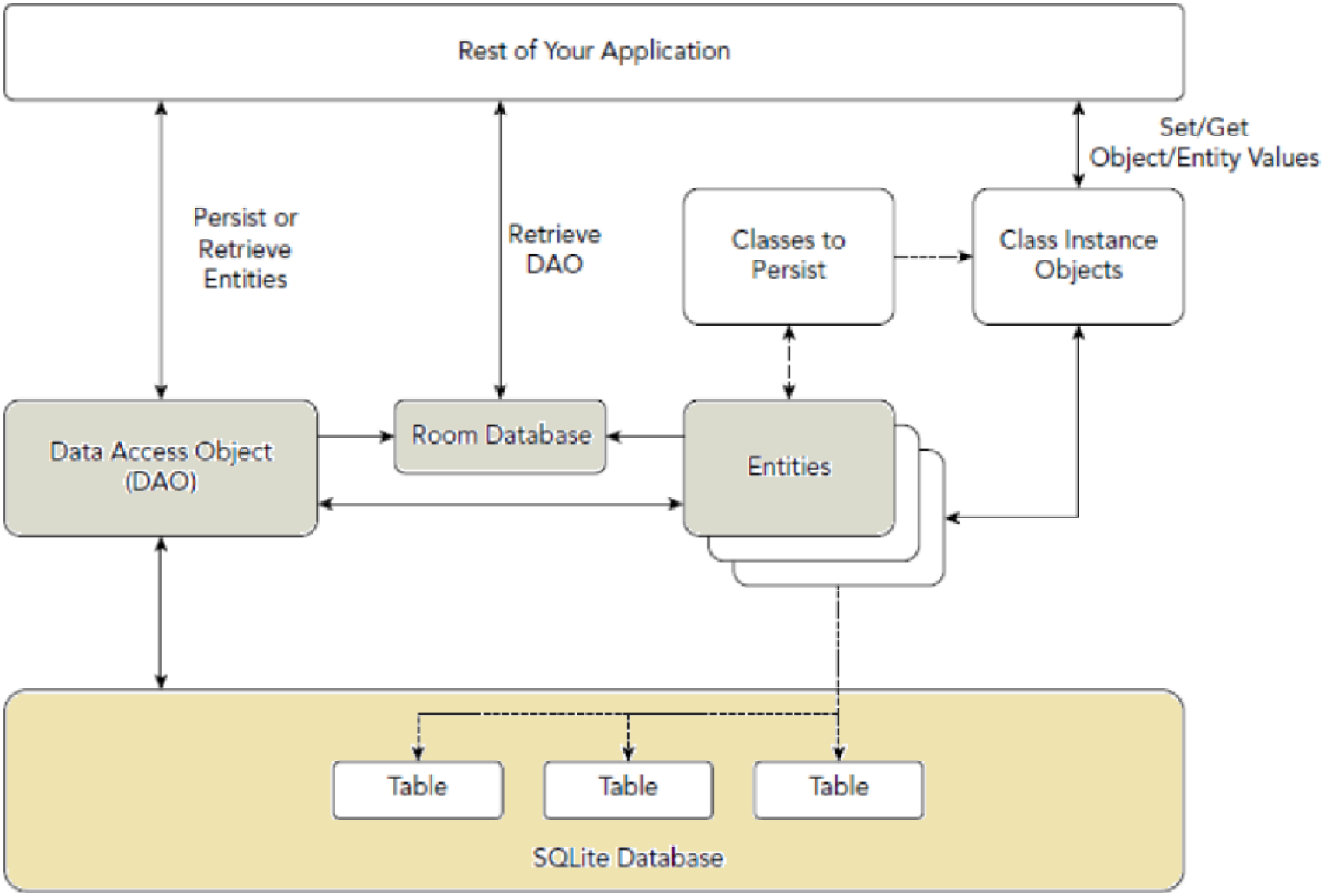


FIGURE 9-2

Defining a Room entity

```
@Entity
data class Schedule(
    @PrimaryKey val id: Int,
    @NonNull @ColumnInfo(name = "stop_name") val stopName: String,
    @NonNull @ColumnInfo(name = "arrival_time") val arrivalTime: Int
)
```

Defining a Room database

- Once your entities are defined, create a new abstract class that extends RoomDatabase, annotating it with a @Database annotation that **includes a list of each of your entity classes** and the current version number:

```
@Database(entities = arrayOf(Schedule::class), version = 1)
abstract class AppDatabase: RoomDatabase() {
    abstract fun scheduleDao(): ScheduleDao
    companion object {
            @Volatile
            private var INSTANCE: AppDatabase? = null
```

Defining a Room database

```
        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context,
                    AppDatabase::class.java,
                    "app_database")
                    .createFromAsset("database/bus_schedule.db")
                    .build()
                INSTANCE = instance

                instance
            }
        }
    }
}
```

Defining Dao

```
/**
 * Provides access to read/write operations on the schedule table.
 * Used by the view models to format the query results for use in the UI.
 */
@Dao
interface ScheduleDao {

    @Query("SELECT * FROM schedule ORDER BY arrival_time ASC")
    fun getAll(): Flow<List<Schedule>>

    @Query("SELECT * FROM schedule WHERE stop_name = :stopName
ORDER BY arrival_time ASC")
    fun getByStopName(stopName: String): Flow<List<Schedule>>

}
```

Using Data Access Objects

- Data Access Objects (DAO) are classes **used to define your Room database interactions**, including methods used to **insert**, **delete**, **update**, and **query** your database.
- If your database includes multiple tables, it's best practice to have **multiple DAO classes, one for each table**.
- DAO's are **defined either as interfaces or abstract classes**, annotated using the @Dao annotation as shown here:

```
@Dao
interface ScheduleDao {
}
```

Using Data Access Objects

- Make DAO's available to your app by adding a new abstract public method to the Room Database class that returns the new DAO:

```
@Database(entities = arrayOf(Schedule::class),
version = 1)
abstract class AppDatabase: RoomDatabase() {
    abstract fun scheduleDao(): ScheduleDao
```

- Within your DAO, create new methods to support each of your database interactions using the @Insert, @Update, @Delete, and @Query annotations.

Inserting Entities

- Use the @Insert annotation to annotate methods that will be used to insert a new object/entity instance into your database.
  - Each **insert method can accept one or more parameters** (including collections), of the type/entity represented by this DAO.

```
@Insert
fun insertAll(vararg users: User)
```

Updating Entities

- You can create methods that update objects stored within your database using the @Update annotation.
  - Each update method can accept one or more entity parameters (including collections).
  - Each object parameter passed in will be matched against the primary key of existing database entities and updated accordingly.

```
@Update
    fun updateUsers(vararg users: User)
```

Deleting Entities

- Use the @Delete annotation to create delete methods.
- Room will use the primary key of each received parameter to find entities within the database and remove them.

```
@Delete
    fun delete(user: User)
```

Querying a Room Database

- The **@Query** annotation allows you to perform read/write operations on the database using SELECT, UPDATE, and DELETE SQL statements, which will be executed when the associated method is called:

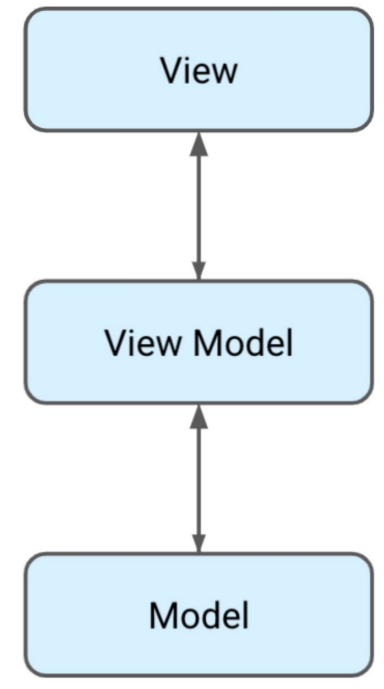
```
@Query("SELECT * FROM user")
fun loadAllUsers(): Array<User>
```

- Each @Query SQL statement is **verified at compile time**.
- To **use method parameters** within the SQL query statement, you can reference them **by prepending a colon (:) to the parameter name**:

```
@Query("SELECT * FROM user WHERE age > :minAge")
fun loadAllUsersOlderThan(minAge: Int): Array<User>
```

Defining the ViewModel

- It's considered best practice to separate the part of the DAO you expose to the view into a separate class called a **view model**.
- This is a **common architectural pattern in mobile apps**.
- Using a view model helps enforce a clear separation between the code for your app's UI and its data model:



Defining the ViewModel

- The ViewModel class is used to **store data related to an app's UI**, and is also **lifecycle aware**, meaning that it responds to lifecycle events much like an activity or fragment does.
  - If lifecycle events such as screen rotation cause an activity or fragment to be destroyed and recreated, the associated ViewModel won't need to be recreated.
- This is not possible with accessing a DAO class directly, so it's **best practice to use ViewModel subclass to separate the responsibility** of loading data from your activity or fragment.

Defining the ViewModel

```
class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel() {

    fun fullSchedule(): Flow<List<Schedule>> = scheduleDao.getAll()

    fun scheduleForStopName(name: String): Flow<List<Schedule>> =
scheduleDao.getByStopName(name)
}

class BusScheduleViewModelFactory(
    private val scheduleDao: ScheduleDao
) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(BusScheduleViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return BusScheduleViewModel(scheduleDao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Creating the ListAdapter

```
class BusStopAdapter(
    private val onItemClicked: (Schedule) -> Unit
) : ListAdapter<Schedule, BusStopAdapter.BusStopViewHolder>(DiffCallback) {

    companion object {
        private val DiffCallback = object : DiffUtil.ItemCallback<Schedule>() {
            override fun areItemsTheSame(oldItem: Schedule, newItem: Schedule):
Boolean {
                return oldItem.id == newItem.id
            }

            override fun areContentsTheSame(oldItem: Schedule, newItem: Schedule):
Boolean {
                return oldItem == newItem
            }
        }
    }
```

Creating the ListAdapter

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): BusStopViewHolder {
    val viewHolder = BusStopViewHolder(
        BusStopItemBinding.inflate(
            LayoutInflater.from( parent.context),
            parent,
            false
        )
    )
    viewHolder.itemView.setOnClickListener {
        val position = viewHolder.adapterPosition
        onItemClicked(getItem(position))
    }
    return viewHolder
}

override fun onBindViewHolder(holder: BusStopViewHolder, position: Int) {
    holder.bind(getItem(position))
}
```

Creating the ListAdapter

```
class BusStopViewHolder(
        private var binding: BusStopItemBinding
    ): RecyclerView.ViewHolder(binding.root) {
        @SuppressLint("SimpleDateFormat")
        fun bind(schedule: Schedule) {
            binding.stopNameTextView.text = schedule.stopName
            binding.arrivalTimeTextView.text = SimpleDateFormat(
                "h:mm a").format(Date(schedule.arrivalTime.toLong() * 1000)
            )
        }
    }
}
```

Responding to data changes using Flow

- You can take advantage of a Kotlin feature called **asynchronous flow** (often just called **flow**) that will allow the DAO to continuously emit data from the database.
- If an item is inserted, updated, or deleted, the result will be sent back to the fragment.
- Using a function called **collect()**, you can call **submitList()** using the new value emitted from the flow so that your ListAdapter can update the UI based on the new data.

Responding to data changes using Flow

```
/**
 * Provides access to read/write operations on the schedule table.
 * Used by the view models to format the query results for use in the UI.
 */
@Dao
interface ScheduleDao {

    @Query("SELECT * FROM schedule ORDER BY arrival_time ASC")
    fun getAll(): Flow<List<Schedule>>

    @Query("SELECT * FROM schedule WHERE stop_name = :stopName
ORDER BY arrival_time ASC")
    fun getByStopName(stopName: String): Flow<List<Schedule>>

}
```

Responding to data changes using Flow

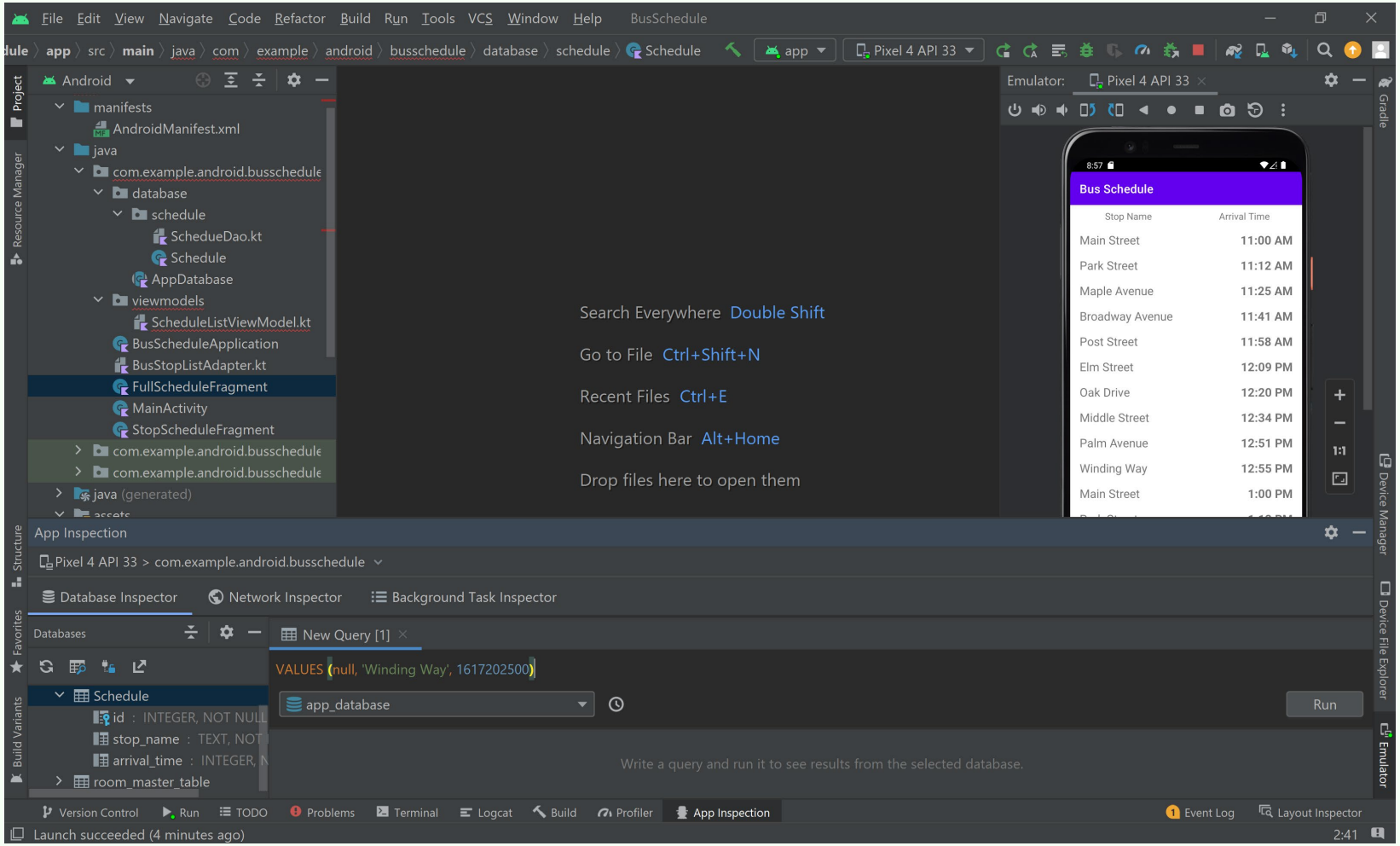
- Functions **getAll** and **getByStopName**() in ScheduleDao return **Flow**<List<Schedule>>.
- The functions **fullSchedule**() and **scheduleForStopName**() in the view model that access the DAO also return **Flow**<List<Schedule>>.
- The suspend function **fullSchedule**() in **FullScheduleFragment**, needs to be called from a coroutine:

```
lifecycle.coroutineScope.launch {
    viewModel.fullSchedule().collect() {
        busStopAdapter.submitList(it)
    }
}
```

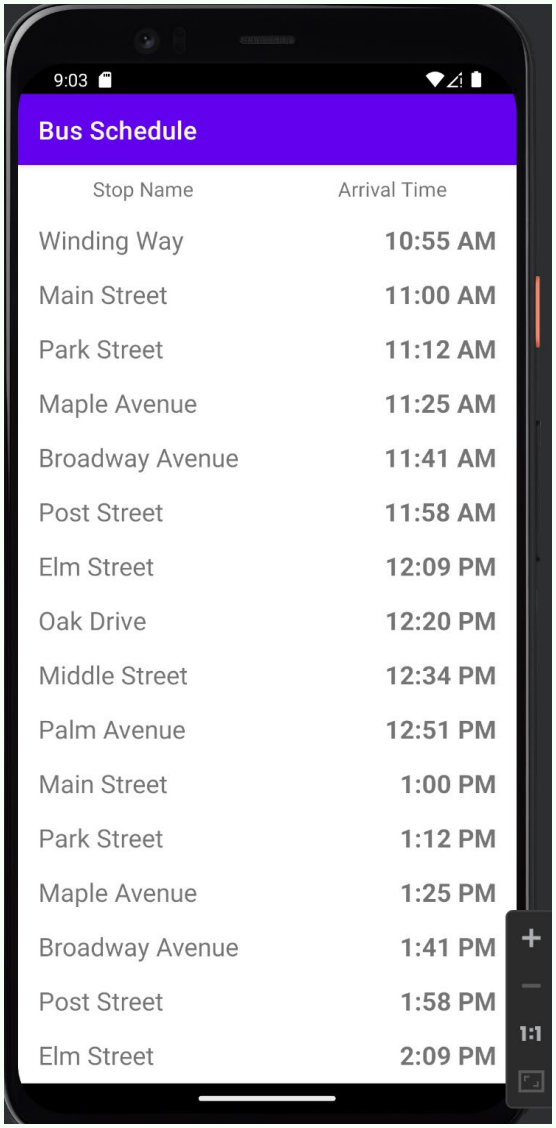
- The same for **scheduleForStopName**() in **StopScheduleFragment**:

```
lifecycle.coroutineScope.launch {
    viewModel.scheduleForStopName(stopName).collect() {
        busStopAdapter.submitList(it)
    }
}
```

Running the BusSchedule App



Running the App



References

- Textbook
- https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow#0